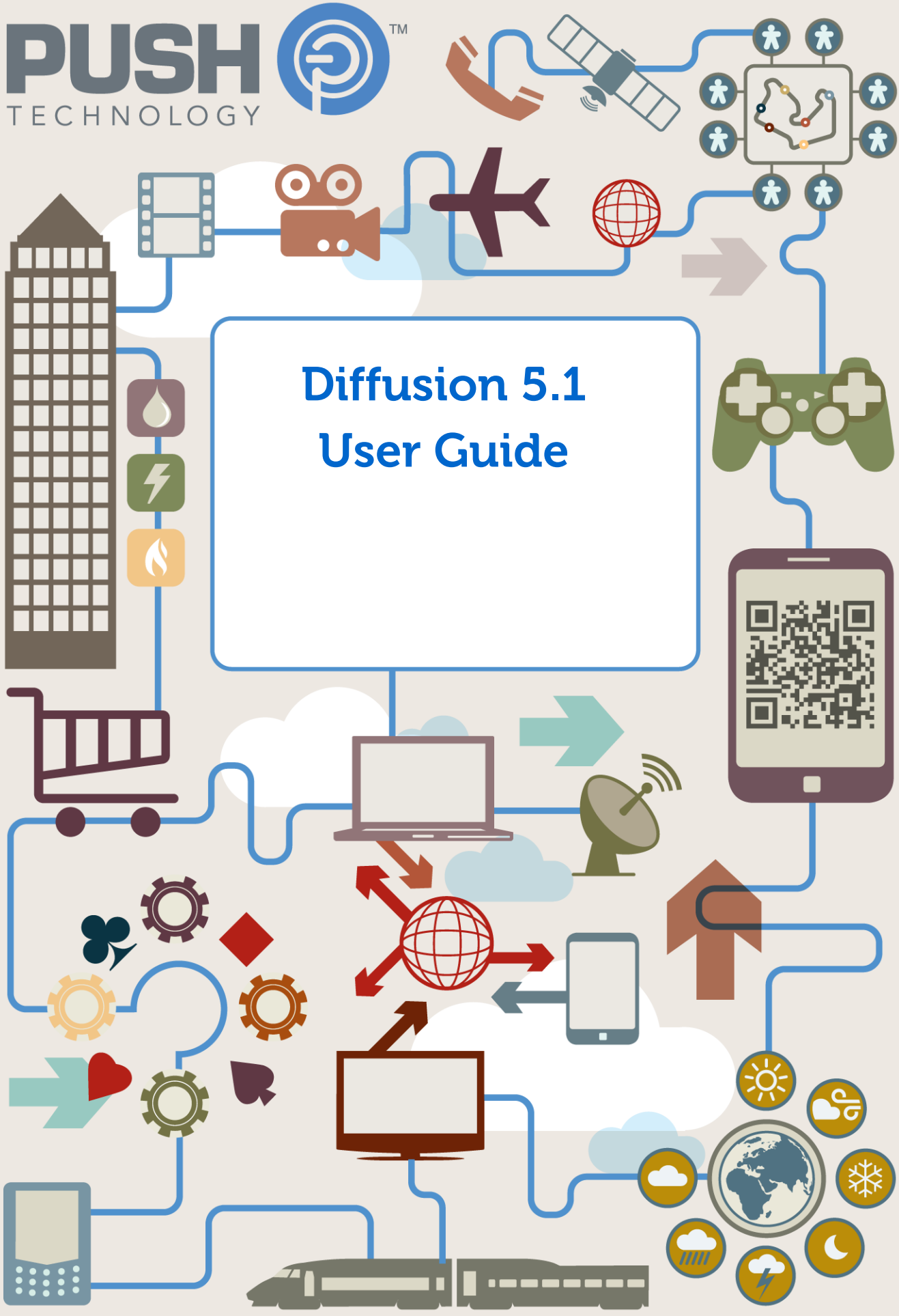
PUSH TECHNOLOGY™

Diffusion 5.1 User Guide



Contents

List of Figures 11

List of Tables 15

Part I: Introduction 19

What's new in Diffusion 5.1? 20

What's new in Diffusion 5.0? 21

Chapter 1: Upgrading 23

Diffusion releases 24

Support and upgrade policy 24

Interoperability 25

Upgrading from version 4.x to version 5.0 26

Upgrading from version 5.0 to version 5.1 29

Upgrading to a new patch release 32

Known issues in Diffusion 5.1 22

Chapter 2: Overview 34

Architecture 35

Data distribution 37

Diffusion server 38

Publishers 39

Topics 40

Clients 41

Control clients 42

Diffusion APIs 42

Part II: Diffusion Guide 44

Chapter 1: Installing 45

System requirements for the Diffusion server 46

Obtaining a Diffusion license 47

Installing the Diffusion server 47

Installing Diffusion using the headless installer 48

Installing Diffusion using Red Hat Package Manager 49

Updating your license file 49

Installed files 50

Verifying your installation 51

Web server installation 53

Chapter 2: Server 56

Server basics 57

Starting the server 57

Running from within a Java application 58

Concurrency....60
Connectors....62
Load balancers....64
Replication....67
Session replication....67
Topic replication....69
Failover of active update sources....70
Configuring replication....72
Configuring your datagrid provider....73

Chapter 3: Web server....75
Interaction with publishers....76
Security....77

Chapter 4: Publishers....78
Publisher basics....79
Defining publishers....79
Loading publisher code....80
Programmatically loading publishers....80
Starting and stopping publishers....80
Publisher topics....82
Receiving and maintaining data....83
Publishing and sending messages....83
Publisher notifications....84
Client handling....86
Publisher properties....86
Using concurrent threads....86
Publisher logging....86
Server connections....86
General utilities....87
Writing a publisher....87
Creating a Publisher class....88
Publisher startup....88
Data state....88
Data inputs....89
Handling client subscriptions....90
Publishing messages....91
Topic locking....91
Handling clients....92
Publisher closedown....92
Testing a publisher....93
Client queues....93
Queue enquiries....93
Maximum queue depth....94
Queue notification thresholds....94
Tidy on unsubscribe....94
Filtering queued messages....95
Client validation....95
Client validation policy types....95
Automatic or manual policies....95
Client validation criteria....96
Client connection validation policy....96
Client subscription validation policy....97
Using policies programmatically....97
Client Geo and WhoIs information....97
The Diffusion WhoIs service....98
Client groups....99

Client notifications....100
Adding a ClientListener....101
Using DefaultClientListener....101
Design patterns....102
Data models....102
Topic structure....103

Chapter 5: Clients....104
Connection....105
Connection details in the Classic API....105
Connecting through an HTTP proxy....106
Topics....107
Messages....107
Client types....107
External clients....108
JavaScript clients....108
Silverlight clients....109
Flash clients....109
WebSocket clients....110
Publisher clients....110
Android clients....110
iOS clients....110
Client close reason....110
Cross domain....112
Protocols supported....115
Interfaces supported....116
Browsers supported....118
Browser limitations....118
WebSocket limitations....119
Cross-origin resource sharing limitations....120
Browser connection limitations....122

Chapter 6: Topics....126
Topic basics....127
Topic tree....130
Topic naming....133
Topic aliasing....134
Creating topics....134
Topic subscription....136
Forced subscription....140
Pre-emptive subscription....141
Topic selection....142
Topic selectors in the Unified API....142
Topic selectors in the Classic API....146
Topic loading....148
Topic data....152
Working with topic data....153
Publishing topic data....154
Paged topic data....167
Routing topic data....174
Child list topic data....176
Topic notify topic data....176
Topic fetch....180
Topic sets....183
Topic attachments....183

Chapter 7: Message conflation....184
Advantages of message conflation....185
Types of message conflation....185
How does conflation work?....187
Configuring conflation....188
Conflation counts....192

Chapter 8: Messages....193
Introduction....194
Message types....195
Creating messages....197
Populating messages....198
User headers....200
Reading messages....200
Records....203
Byte encoding of a message....206
Character encoding....209
Message priority....209
Acknowledged messages....210
Fragmented messages....213
Message filters....214
Metadata....216
Metadata in the Classic API....217

Chapter 9: Event publishers (deprecated)....223
Event publishers that use the Classic API....224

Chapter 10: Control client....226
Advantages of control client....227
Developing a control client....228
Event handling with a control client....229
Maintaining topics from a control client....230
Creating paged topics....231
Removing topics with sessions....233
Updating topics from a control client....234
Building updates for paged topics....235
Managing subscriptions from a control client....236
Managing clients from a control client....237
Messaging from a control client....238
Comparison of remote control and control client....238

Chapter 11: Unified API....240
Advantages of the Unified API....242
Key concepts in the Unified API....242
Sessions....242
Features....244
Callbacks and streams....245
Handlers and listeners....247
Content....247
Topic details....255
Supported client platforms....258
Java....258
.NET....261

C ... 264
Features ... 266
AuthenticationControl ... 267
ClientControl ... 274
Messaging ... 278
MessagingControl ... 283
Pings ... 286
Security ... 290
SubscriptionControl ... 294
TopicControl ... 297
Topics ... 312
TopicUpdateControl ... 325

Chapter 12: Classic APIs ... 342
Table of APIs ... 343
Java API ... 345
Client API ... 346
(Deprecated) Event publisher API ... 349
.NET API ... 350
Client API ... 350
(Deprecated) Event publisher API ... 352
JavaScript API ... 354
Using the JavaScript API ... 354
ActionScript API ... 360
Using the ActionScript API ... 360
Reconnecting with the ActionScript API ... 363
Silverlight API ... 364
Using the Silverlight API ... 365
iOS API ... 368
DFClient ... 369
Installing the docset ... 372
Android API ... 372
Using the Android API ... 372
C API ... 376
Using the C API ... 376
diffusion-wrapper.js ... 377
How to use Diffusion wrapper ... 378

Chapter 13: System management ... 380
Going to production ... 381
General management ... 382
Classic deployment ... 382
Hot deployment ... 382
What's in a DAR file? ... 383
Building a DAR file ... 384
Deployment methods ... 385
Using JMX ... 386
MBeans ... 386
Using Visual VM ... 394
Using JConsole ... 397
Statistics ... 400
Configuring statistics ... 401
Diffusion monitoring console ... 402
Basic integration with Splunk ... 411

Chapter 14: Security ... 414

User access control ... 415
Authentication ... 415
Authentication handlers ... 418
Authorization handlers ... 430
Network security ... 432

Chapter 15: Distribution ... 435
Publisher clients ... 436
Distributed topics ... 439
Distribution examples ... 439

Chapter 16: Configuration ... 441
XML configuration ... 442
server ... 444
connectors ... 456
publishers ... 461
web-servers ... 465
logs ... 470
management ... 472
replication ... 473
statistics ... 475
connection-validation-policies ... 478
subscription-validation-policies ... 479
env ... 481
aliases ... 481
mimes ... 482
Additional XML files ... 482
Programmatic configuration ... 482
Using the configuration API ... 483
The configuration tree ... 485

Chapter 17: Adapters ... 486
JMS Adapter ... 487
Installing the JMS adapter ... 488
Configuring the JMS Adapter ... 488
JMS Adapter examples ... 491
Receiving messages from JMS ... 492
Sending messages to JMS ... 492
Processing a request-reply message with a Diffusion client ... 493
Sending a request-reply message from a Diffusion client ... 494

Chapter 18: Tuning ... 496
Buffer sizing ... 497
Message sizing ... 498
Client queues ... 499
Client multiplexers ... 499
Write selectors ... 500
Connectors ... 500
Thread pools ... 501
Client reconnection ... 504
Client failover ... 506
Client throttling ... 508
Memory considerations ... 509
Garbage collection (Java HotSpot VM) ... 509
Platform-specific issues ... 510

Socket issues....510
Increasing number of open files....511
Publisher design....511

Chapter 19: Diagnostics....513
Logging....514
Logging API....517
Connection counts....518
Message diagnostics....519
JavaScript diagnostics....520
Flex and Flash diagnostics....523
Windows diagnostics....524
Debugging a publisher....525
Log Messages....528

Chapter 20: Introspector....557
Supported platforms....558
Installing from update site....558
Installing subsequent plugin updates....561
Uninstalling....562
Opening the Diffusion perspective....563
Adding servers....565
Opening servers....566
Exploring the topics....566
Getting topic values....566
Configuring columns....568
Ping servers....568
Count topics....568
Using the clients view....568
Ping....569
Statistics....570
Topics....570
Logging....570
Server logs....570
Property obfuscator....571

Chapter 21: Demos....572
Demos....573
Building the demos using mvndar....573

Chapter 22: Testing....575
Flex/Flash client....576
Java client test tool....579
Java event publisher test tool....583
JavaScript client test tool....585
Silverlight client test tool....586
Windows client test tool (.NET)....588
Windows event publisher test tool (.NET)....591
Stress test tuning....593
Stress test....594
Benchmarking suite....595
Test tools....595

Chapter 23: Tools....597

Tools for Amazon Elastic Compute Cloud (EC2) ... 598
Tools for Joyent ... 599
Using Maven to build Java Diffusion applications ... 600
Building client applications with Maven ... 600
Building publishers and other server application code with Maven ... 601

Chapter 24: Appendix: Protocol ... 603
Protocols overview ... 604
DPT ... 604
HTTP protocol ... 611
WebSocket protocol ... 615
Command protocols ... 617

Appendices ... 622

Appendix A: Document conventions ... 623

Appendix B: Glossary ... 624
A ... 625
C ... 625
D ... 626
E ... 626
F ... 626
H ... 627
I ... 627
L ... 627
M ... 627
P ... 627
Q ... 628
R ... 628
S ... 628
T ... 629
U ... 630

Appendix C: Trademarks ... 631

Appendix D: Copyright Notices ... 633
Apache Commons Codec ... 634
CocoaAsyncSocket ... 634
cron4j ... 634
d3 ... 634
FastColoredTextBox ... 635
Fluent validation ... 635
GeoIP API ... 635
GeoLite City Database ... 635
geronimo-jms_1.1_spec ... 635
Hazelcast ... 635
HPPC ... 636
htmlcompressor ... 636
IKVM ... 636
JCIP Annotations ... 637
JCommon ... 637
jQuery ... 637

JSON....637
json-simple....637
JZlib....637
Knockout....637
Metrics....638
opencsv....638
Picocontainer....638
Protocol Buffers....638
Rickshaw....638
SLF4J....639
SocketRocket....639
SWT 2D Graphics....639
Tapestry (Plastic)....639
TrueLicense....640
Licenses....640

List of Figures

Figure 1: Basic architectural components....35
Figure 2: A scalable, resilient architecture....36
Figure 3: Components in a Diffusion server....38
Figure 4: Topics....40
Figure 5: Thread diagram....62
Figure 6: Sticky-IP in F5 BIG-IP....65
Figure 7: Information sharing using a datagrid....67
Figure 8: Session replication....67
Figure 9: Topic replication....69
Figure 10: Failover of an active update source....71
Figure 11: The message queue....93
Figure 12: Flow of requests and responses when connecting to Diffusion through a proxy....106
Figure 13: Using a load balancer to composite two URL spaces into one....114
Figure 14: Flat structure....130
Figure 15: Hierarchical topic structure....131
Figure 16: Topic aliasing....134
Figure 17: Message flow without conflation enabled....185
Figure 18: Message flow with simple replace conflation enabled....186
Figure 19: Message flow with simple append conflation enabled....186
Figure 20: Message flow with merge and replace conflation enabled....186
Figure 21: Session state model....244
Figure 22: A callback....245
Figure 23: A stream....246
Figure 24: XCode documentation browser....369
Figure 25: Diffusion wrapper....378

Figure 26: Example folder structure inside a DAR file....383
Figure 27: The server MBean stopController operation showing in JConsole....386
Figure 28: Java VisualVM: Overview tab....395
Figure 29: Java VisualVM: Monitor tab....395
Figure 30: Java VisualVM: Threads tab....396
Figure 31: Java VisualVM: Profiler tab....396
Figure 32: JConsole New Connection dialog: Local Process....398
Figure 33: Tuning and monitoring in JConsole....398
Figure 34: JConsole New Connection dialog: Remote Process....400
Figure 35: The default console layout....403
Figure 36: The table of publishers....404
Figure 37: Publisher statistics graphs....404
Figure 38: The table of topics....405
Figure 39: Details of the topic publishing the CPU load of the host server....406
Figure 40: The table of clients....406
Figure 41: The table of log entries....407
Figure 42: Editing the Access Policy....407
Figure 43: Notification that the Diffusion server has stopped....408
Figure 44: The default Diffusion Details panel....408
Figure 45: Editing the properties of the Diffusion Details panel....409
Figure 46: Visualizing the CPU load on a server at a specific time....410
Figure 47: Editing and adding to the set of topics for this panel....410
Figure 48: Welcome tab of the Splunk web UI....412
Figure 49: The Splunk Set source type dialog....412
Figure 50: The Data Preview panel....413
Figure 51: The Splunk search summary panel....413
Figure 52: Authentication process for clients....416
Figure 53: A composite authentication handler....419
Figure 54: Distributors....440

Figure 55: Aggregators....440
Figure 56: JMS adapter topic tree layout....487
Figure 57: Subscription flow....492
Figure 58: Sending flow from a Diffusion client to a JMS topic (or queue)....493
Figure 59: Request-reply initiated by a JMS client and serviced by a Diffusion client....494
Figure 60: Request-reply initiated by a Diffusion client and serviced by a JMS client....495
Figure 61: Reconnection scenario....506
Figure 62: Normal and throttled client queues....509
Figure 63: Firefox Console....520
Figure 64: Chrome's console....521
Figure 65: Internet Explorer console....522
Figure 66: Opera console....522
Figure 67: Safari's console....523
Figure 68: New Java Project....526
Figure 69: Creating a new Java class....527
Figure 70: Example classpath entries....528
Figure 71: Adding a repository....558
Figure 72: Install dialog....559
Figure 73: Accept the license agreement....560
Figure 74: Click OK....561
Figure 75: Restarting....561
Figure 76: About Eclipse dialog....562
Figure 77: Installed plugins....563
Figure 78: Perspective....564
Figure 79: Views....564
Figure 80: Add a server....565
Figure 81: Edit server details....566
Figure 82: View topic values....567

Figure 83: Re-order columns....568
Figure 84: Ping a server....568
Figure 85: Topic count....568
Figure 86: Ping clients....569
Figure 87: Server log entries....570
Figure 88: Property Obfuscator dialog....571
Figure 89: Flex client: Connection tab....576
Figure 90: Flex client: Send tab....577
Figure 91: Flex client: Messages tab....578
Figure 92: Flex client: Log tab....579
Figure 93: External client tester: Connection tab....580
Figure 94: External client tester: Send tab....581
Figure 95: External client tester: Messages tab....582
Figure 96: External client tester: Message details window....583
Figure 97: Event publisher test tool: Send tab....584
Figure 98: Event publisher test tool: Messages tab....585
Figure 99: JavaScript test tool....586
Figure 100: Silverlight test tool: Connection tab....586
Figure 101: Silverlight test tool: Send tab....587
Figure 102: Silverlight test tool: Messages tab....588
Figure 103: Windows client test tool: Connection tab....589
Figure 104: Windows client test tool: Send tab....590
Figure 105: Windows client test tool: Messages tab....591
Figure 106: Windows event publisher test tool: Send tab....592
Figure 107: Windows event publisher test tool: Messages tab....593

List of Tables

Table 1: API features removed in version 5.0 27
Table 2: API features deprecated in version 5.0 27
Table 3: API features removed in version 5.1 30
Table 4: API features deprecated in version 5.1 30
Table 5: Installed files 50
Table 6: Tools and utilities 51
Table 7: Connectors properties 63
Table 8: Connection restrictions 63
Table 9: Routing strategies 66
Table 10: Start publisher 81
Table 11: Stop publisher 81
Table 12: Notification methods 84
Table 13: General publisher utilities 87
Table 14: Validation criteria 96
Table 15: WhoIs 97
Table 16: WhoIs service 98
Table 17: Client listener notifications 100
Table 18: Client types 107
Table 19: Supported protocols by client 116
Table 20: Tier 1 supported client platforms 117
Table 21: Supported browsers 118
Table 22: Browser Plugins 118
Table 23: Internet Explorer support for WebSocket 119
Table 24: Firefox support for WebSocket 119
Table 25: Chrome support for WebSocket 119

Table 26: Safari support for WebSocket..... 119
Table 27: Opera support for WebSocket..... 120
Table 28: iOS support for WebSocket..... 120
Table 29: Android support for WebSocket..... 120
Table 30: Internet Explorer support for CORS..... 121
Table 31: Firefox support for CORS..... 121
Table 32: Chrome support for CORS..... 121
Table 33: Safari support for CORS..... 121
Table 34: Opera support for CORS..... 121
Table 35: iOS support for CORS..... 122
Table 36: Android support for CORS..... 122
Table 37: Internet Explorer maximum supported connections..... 122
Table 38: Firefox maximum supported connections..... 123
Table 39: Chrome maximum supported connections..... 123
Table 40: Safari maximum supported connections..... 123
Table 41: Opera maximum supported connections..... 123
Table 42: iOS maximum supported connections..... 124
Table 43: Android maximum supported connections..... 124
Table 44: Restricted characters..... 133
Table 45: Methods for adding topics to publishers..... 135
Table 46: Types of topic selector..... 143
Table 47: Descendant pattern qualifiers..... 143
Table 48: Selector examples..... 148
Table 49: Publishing topic data types..... 154
Table 50: Paged topic data types..... 167
Table 51: Duplicates policies..... 168
Table 52: Usable methods with ordered topic data..... 169
Table 53: Usable methods with unordered topic data..... 170
Table 54: Paged topic commands..... 171

Table 55: Paged topic notifications..... 172
Table 56: Page status..... 173
Table 57: Notification levels..... 178
Table 58: Selection modes..... 178
Table 59: Conflation policy elements..... 188
Table 60: Conflation policy modes..... 189
Table 61: Action depending upon merge result..... 190
Table 62: Message format..... 194
Table 63: Separator bytes..... 203
Table 64: Types of byte encoding..... 206
Table 65: Encoding support transports..... 208
Table 66: Creating a fragmented message..... 213
Table 67: Fragmented message lifecycle..... 213
Table 68: Data types..... 217
Table 69: Data types for metadata fields..... 248
Table 70: Tier 1 supported client platforms..... 258
Table 71: Java interfaces..... 258
Table 72: .NET interfaces..... 261
Table 73: C functions..... 264
Table 74: Matrix of supported features by language..... 267
Table 75: Feature matrix..... 344
Table 76: Java APIs..... 345
Table 77: Connection types..... 347
Table 78: Types of connection that can be specified from the .NET client..... 350
Table 79: JavaScript functions called on events..... 355
Table 80: Client operations that require authentication..... 417
Table 81: Types of authentication handler..... 419
Table 82: Classes and interfaces in c.p.d.client.security.authentication..... 429
Table 83: Interfaces in c.p.d.client.types..... 429

Table 84: Authorization handler methods..... 430
Table 85: Client security.....432
Table 86: XML Value types..... 442
Table 87: Properties that can be specified when configuring the JMS adapter...490
Table 88: Values that can be configured for a thread pool..... 501
Table 89: Events that a thread pool notification handler can act on.....502
Table 90: Log levels..... 514
Table 91: Fields included in the logs.....515
Table 92: Mapping between LogWriter methods and Diffusion log levels..... 518
Table 93: Location of the flashlog.txt file..... 524
Table 94: Location of the policyfiles.txt file.....524
Table 95: Client properties in the Eclipse client view.....569
Table 96: Demos provided with the Diffusion server.....573
Table 97: Tuning changes for stress testing.....593
Table 98: Testing tools..... 595
Table 99: Targets..... 598
Table 100: Properties for targets start, stop and status..... 599
Table 101: Additional properties for targets deploy and undeploy..... 599
Table 102: Artifacts..... 600
Table 103: Message types..... 608
Table 104: Key.....609
Table 105: Message interactions.....610
Table 106: Commands (Message type 36)..... 619
Table 107: Notifications (Message type 41)..... 619
Table 108: Topic types.....620
Table 109: Topic properties..... 620
Table 110: Typographic conventions used in this manual.....623

Part

I

Introduction

In this section:

- What's new in Diffusion 5.1?
- What's new in Diffusion 5.0?
- Upgrading
- Known issues in Diffusion 5.1
- Overview

Welcome to the Push Technology User Manual for Diffusion™

Diffusion is a development software platform that removes the complexity and associated challenges of developing for scale, coping with the explosion of data across networks, delivering a rich application experience and real-time conversational interactions

Deployed in your organization's own data centers or in the cloud, Diffusion gives developers the toolkit to create high performance, value driven and reliable web and mobile applications.

Diffusion is able to intelligently understand data to automatically remove out of date or redundant data. As a result, you can use it to efficiently distribute fast-changing data to a large number of simultaneously connected users.

This offers a rich application experience for end users as they receive only relevant and up to date data. It also enables you to reduce the amount of infrastructure and bandwidth consumption required to distribute data to web and mobile applications on any internet connected device.

The manual is regularly updated, but if you require further help, please contact the team at support@pushtechnology.com

What's new in Diffusion 5.1?

The latest version of Diffusion contains new functions, performance enhancements and bug fixes.

Key features

A complete list of the latest updates to Diffusion can be found in the Release Notes available at http://download.pushtechnology.com.

Paged topic support in the Java™ and .NET Unified API

You can use the TopicControl feature to create paged record topics and paged string topics and to define rule-based comparators to use for ordering the lines in the paged topic. For more information, see Creating a paged topic.

You can use the TopicUpdateControl feature to update paged record topics and paged string topics. For more information, see Building an update for a paged topic.

UpdateSource capabilities in the Unified API

The UpdateSource capabilities replace the TopicSource capabilities. UpdateSource includes the ability to build more complex updates, support for more topic types, and better handling of unexpected closes.

For more information, see Upgrading from 5.0 to 5.1 or the API documentation.

Remove topics after the control client closes using the Unified API

The TopicControl feature now enables you to specify whether to remove sections of the topic tree after a control client session closes.

For more information, see Removing topics with sessions on page 233.

.NET and C Unified API production support

The .NET and C Unified API are now supported for production use. Both APIs contain functionality to implement a control client.

For more information, see .NET on page 261 and C on page 264.

Proxy support

Clients that use the Java Unified API can now connect to the Diffusion server through a proxy. The Unified API enables you to connect through the proxy unauthenticated, with basic authentication, or through any other authentication process by implementing your own challenge handlers.

For more information, see Connecting through an HTTP proxy on page 106.

WebSocket support in the iOS® Classic API

Clients implemented using the iOS Classic API can now connect to the Diffusion server through the WebSocket protocol.

Streams replace listeners for receiving content through the Unified API

The `Listeners` in the Topics and Messaging features are now deprecated. Listeners have been replaced by streams. `Topics.TopicStream` receives topic events, such as topic updates for a topic or topics. `Messaging.MessageStream` receives messages sent through a topic or topics.

Streams provide advantages over listeners as a stream has a logical end. A stream can be closed or discarded, at which time the stream has the opportunity to do any required cleanup or take any required actions.

Flow control (Java Unified API)

The Java client library can now control the flow of requests from a client to decrease the likelihood of the client's outbound queue or the client queue on the server overflowing and causing the client to be disconnected.

This process happens automatically when the client detects conditions that might cause a queue overflow and increases the reliability of the Java client.

Related Links

Upgrading on page 23
If you are planning to move from an earlier version of Diffusion to version 5.1, review the following information about changes between versions.

What's new in Diffusion 5.0?

The latest version of Diffusion contains new functions, performance enhancements and bug fixes.

Key features

A complete list of the latest updates to Diffusion can be found in the Release Notes available at http://download.pushtechnology.com.

New high availability features

In version 5.1, Diffusion introduces the following new high availability features: session replication, topic replication, and failover of the active update source. These features use a datagrid to share data between multiple Diffusion servers.

Session replication shares client session information between servers. If a client loses connection to a server, it is reconnected through a load balancer to another server that has access to all of the client's session information.

Topic replication shares topic information – such as the topic definition and metadata – and topic data between servers. If a server becomes unavailable, the topic information and data is available on another server.

Only one server can act as the active update source for a topic or branch of the topic tree. If that server becomes unavailable, other servers can take over as the active update source for those topics.

For more information, see Replication on page 67.

Control client

Control clients are a way to package application logic that controls a Diffusion server. Unlike publishers, control clients run as a separate process outside of the server and use the Diffusion client library to communicate with the server.

Control clients use the Unified API to provide a secure remote control experience that can use all of the supported protocols to communicate with the Diffusion server. They can be implemented in any of the supported languages. For more information, see Control client on page 226.

Introducing the Unified API

Beginning in version 5.0, Diffusion is transitioning to a new public API. The Unified API will make available the capabilities of standard clients, control clients, and event publishers in one consistent, modular interface. For more information, see Unified API on page 240.

For version 5.1, the control features are now available. This enables you to replace remote control with the richer experience of control client.

The Classic API (the API used in version 4 and earlier) is still supported in 5.1 for clients and event publishers. The remote control API is no longer supported.

Improved performance

Diffusion can now serve up to 150% more messages per second to 60% more clients by using a new queuing mechanism.

In benchmark tests, using 50 topics and 125-byte messages, Diffusion served 15 million messages per second to 87,000 clients. Diffusion used 24 threads and three client processes to achieve this performance.

New authentication model

In Diffusion 5.1 we have split out the concept of authentication from that of authorization. You can write and configure both remote and local authentication handlers.

In previous versions, the authentication capability was provided by authorization handlers. Using authorization handlers for authentication is now deprecated. We recommend that you re-implement your authentication logic using the version 5 authentication APIs.

For more information, see User access control on page 415.

JavaScript® API for paged topics

The JavaScript API now includes improved capability to work with paged topics. For more information, see Paged topic data in JavaScript on page 358.

Iframe streaming

Iframe streaming connections are now available over the HTTP protocol. For more information, see Iframe streaming on page 615.

Liveness monitoring in Flex® and JavaScript

The Flex and JavaScript client libraries now include liveness monitors that listen for activity from the server and raise an event if the lack of activity indicates that the connection has been lost. This enables the client to reconnect in the event of a lost connection.

For more information, see Reconnecting with the ActionScript API on page 363 and Reconnecting with the JavaScript API on page 357

Known issues in Diffusion 5.1

Be aware of the following issues when using Diffusion 5.1.

.NET Unified API: the wrong error reason might be passed to `onError()` callbacks

In Diffusion 5.1, new Unified API methods use callback interfaces that provide an `onError()` method to handle error conditions. This is an improvement over the `onDiscard()` callback method used in Diffusion 5.0 because it provides an `ErrorReason` argument that allows the application to determine the reason for failure. In Diffusion 5.1.0, the .NET Unified API might provide an incorrect `ErrorReason`. This known issue affects the `ITopicControl.RemoveTopicsWithSession` method of the TopicControl feature and the `ITopicUpdateSource` interface of the TopicUpdateControl feature.

This issue is tracked in support case 10451.

Chapter 1

Upgrading

In this section:

- Diffusion releases
- Support and upgrade policy
- Interoperability
- Upgrading from version 4.x to version 5.0
- Upgrading from version 5.0 to version 5.1
- Upgrading to a new patch release

If you are planning to move from an earlier version of Diffusion to version 5.1, review the following information about changes between versions.

We recommend that you upgrade incrementally through Diffusion versions. For example, if you are upgrading from version 4.x to version 5.1, first follow the upgrade steps from version 4.x to 5.0, then follow the steps to upgrade from version 5.0 to 5.1.

Release notes are available at the following location: http://download.pushtechnology.com

Related Links

What's new in Diffusion 5.1? on page 20

The latest version of Diffusion contains new functions, performance enhancements and bug fixes.

Diffusion releases

Diffusion is an enterprise-class data distribution engine. As such customers have a reasonable expectation of enterprise-class supportability between releases and a smooth upgrade path.

Diffusion releases are numbered according to the following scheme: X.Y.Z_Q. Each of the digits has the following significance:

- X is the major release number. Diffusion major releases incorporate significant new features and functionality over previous major releases. Diffusion major releases are typically made around the lifecycle of a year.
- Y is the minor release number. Diffusion minor releases incorporate bug fixes and minor features and functionality improvements. Diffusion minor releases are typically made around the lifecycle of a quarter.
- Z is the patch release number. Diffusion patch releases only incorporate bug fixes. Patch releases never contain new features. Patch releases are produced on an ad-hoc basis.
- Q is the internal build number and typically not relevant to customers.

Support and upgrade policy

Push Technology endeavors to maintain compatibility between releases wherever possible and to provide users with appropriate notice of any changes that might require the user to update their own code.

Major releases

- Major releases involve significantly new or changed functionality. As such Push Technology does not guarantee compatibility between major releases. Customers might be required to make code or architectural changes to move between major releases of Diffusion. Wherever possible, we provide a reasonable mechanism for upgrade through documentation, tools and training.

Minor releases

- From time to time, Push Technology removes support for Diffusion APIs. When APIs are due for removal – whether methods or classes – they are first deprecated in a prior release. An API supported in release X.Y.Z is deprecated at the earliest in X.Y+1.Z and removed at the earliest in X.Y+2.Z.
- The final minor release of any major release is called the terminal release. The terminal release of any major release is supported for a period of 18 months from its initial release date or a period of 18 months after the next major release has been issued whichever is later. Support in this context includes fixes and patch releases for the entire 18-month period.
- Non-terminal releases are supported for a period of 12 months after the next minor release of the same major release has been issued. After that period customers are required to upgrade to the latest minor release to receive support. However, fixes and patches are only provided for the initial 6 months of this period. After 6 months customers are required to upgrade to the latest minor release to receive fixes and patches.

Patch releases

- Push Technology guarantees binary and source level compatibility between patch releases on the same major/minor version. Upgrading to a new patch release does not require a recompile to either client or server implementations.
- No new features are delivered as part of a patch release – customers requiring enhancements to a particular version of Diffusion are required to upgrade to either a new minor or major version.

- Bug fixes are officially delivered and supported on only a new patch release. Push Technology might give customers patches for particular releases to verify functionality but to be supported customers are required to pick up the next patch release incorporating that fix.
- New patch releases invalidate older patch releases of the same major/minor version. To investigate support issues, customers are required to upgrade to the latest patch release of the major/minor release that they are currently using.
- APIs are not removed, changed or added in any patch release.
- Configuration items are not removed, changed or added in any patch release.

Interoperability

If you plan to use different versions of Diffusion servers and clients together, review the following information that summarizes support between versions.

Interoperation between clients and servers

The following table describes which client versions interoperate with which server versions, through the Classic API or Unified API:

	Server version			
Client version	**4.5**	**4.6**	**5.0**	**5.1**
4.5	YES	YES	YES	YES
4.6	YES	YES	YES	YES
5.0 Classic API	NO	NO	YES	YES
5.0 Unified API	NO	NO	YES	NO
5.1 Classic API	NO	NO	YES	YES
5.1 Unified API	NO	NO	NO	YES

Caution: For the 5.x release, we do not guarantee interoperability between clients and servers that use different 5.x versions of the Unified API. Be aware that when you upgrade between 5.x versions you might have to upgrade all servers and Unified API clients together.

Interoperation between servers

Publishers deployed to Diffusion servers can connect to and communicate with publishers deployed to Diffusion servers of different versions. The following table describes which server versions interoperate:

Server versions	**4.5**	**4.6**	**5.0**	**5.1**
4.5	YES	YES	YES	YES
4.6	YES	YES	YES	YES
5.0	YES	YES	YES	YES
5.1	YES	YES	YES	YES

Related Links

Classic APIs on page 342
Diffusion provides a number of Application Programming Interfaces (APIs) which allow user-written applications to make use of Diffusion.

Unified API on page 240

The Diffusion Unified application programming interface (API) provides a consistent interface to the Diffusion server, whichever role your client application performs: standard client or control client.

Upgrading from version 4.x to version 5.0

Consider the following information when upgrading from Diffusion version 4.x to version 5.0.

Upgrading your applications

Server-side components

Recompile all Java application components that are deployed to the Diffusion server, such as publishers and authorization handlers, against the new version `diffusion.jar` file. This file is located in the `lib` directory of your new Diffusion server installation.

Some features that your Java application components might use have been removed or deprecated. Review the API changes information in the following section to see if these changes affect your applications.

Remote control

The remote control APIs are no longer supported. Reimplement your remote control as a control client using the Unified API control features. For more information, see Control client on page 226 and Unified API on page 240.

Clients

You can choose not to recompile your client applications and continue to use client libraries from a previous release. If you choose to use client libraries from a previous release, ensure that the libraries are compatible with the new server. For more information, see Interoperability on page 25.

You can choose to upgrade your client applications to use the new client libraries. To do this, recompile the client applications against the client libraries located in the `clients` directory of your new Diffusion server installation. Some features that your client applications might use have been removed or deprecated. Review the API changes information in the following section to see if these changes affect your applications.

API changes

Further information about removed or deprecated features is available in following locations:

- The release notes provided in the `docs` directory of your Diffusion installation
- The API documentation located at http://docs.pushtechnology.com/5.0

The following table lists API classes and methods that have been removed. If you attempt to recompile application code that uses these classes or methods against the version 5.0 APIs, it fails. Rewrite your application code to not include these features.

Table 1: API features removed in version 5.0

API affected	Removed feature	Suggested alternative
Java API .NET API	Remote control	Reimplement your remote control applications as control clients using the Unified API. For more information, see Control client on page 226 and Unified API on page 240.
Java API	Methods in the `APIProperties` class: • `setInboundThreadPoolSize` • `getInboundThreadPoolSize`	Use the `ThreadsConfig` class instead. For more information, see Java Unified API documentation.
Android™ API	Methods in the `DiffusionClient` class: • getCredentials • setCredentials	Use the methods in `ServerDetails` or `ConnectionDetails` instead. For more information, see Android Classic API documentation.
Java API	`MessageComparator` interface and `compareTo` and `equals` methods on all Message classes.	Use conflation policies instead. For more information, see see Java Unified API documentation.
Java API	`TopicDetails` class	Use the `TopicDefinition` class instead. For more information, see Java Unified API documentation.
Java API	Methods in the `ThreadsConfig` class: • setWriterSelectors • getWriterSelectors	No longer used and no alternative required.
Java API	`Management`, `Proxy`, and `ServerProxy` interfaces	No longer used and no alternative required.
Java API	`Publisher.consoleLogLevelChange`	No longer used and no alternative required.
Java API	`ThreadServer.getOutboundThreadPool`	

The following table lists API classes and methods that have been deprecated. If your application code uses these classes or methods, consider rewriting your application code to not include these features.

Table 2: API features deprecated in version 5.0

API affected	Deprecated feature	Suggested alternative
Java API	Using authorization handlers for authentication and the `AuthorisationHandler.canConnect` method.	Use authentication handlers instead. For more information, see Authentication handlers on page 418.
Java API	`APIProperties` class	Use methods in the `Utils` or `RootConfig` classes instead.

API affected	Deprecated feature	Suggested alternative
		For more information, see Java Unified API documentation.
Java API .NET API	Event publishers	Reimplement your event publishers as control clients using the Unified API. For more information, see Control client on page 226 and Unified API on page 240.
Java	`Client.getNumberOfMessagesSent` and `Client.getNumberOfMessagesReceived`	Use `Client.getStatistics` instead. For more information, see Java Unified API documentation.
Java	Methods that navigate up from a configuration item to its parent configuration item.	Instead navigate down from the root configuration item. For more information, see Java Unified API documentation.
Java	`MNode.getMessage`	No longer used and no alternative required.

The following list includes behavior that has changed in the API. If your application code relies on the previous behavior, rewrite your application code to take into account the new behavior.

- The Publisher API methods that add topics no longer block until automatic pre-emptive subscriptions have been processed. Matching pre-emptive subscriptions are be completed in the background.
- The Java API now enables you to set auto-subscribe using a `TopicDefinition`
- The format of the generated client IDs has changed
- `getStatistics` no longer returns null if statistics recording is disabled. Instead it returns a value of -1.
- Clients that subscribe to topics that they are already subscribed to, no longer receive an initial topic load.

Upgrading your server installation

To upgrade your Diffusion server installation, complete the following steps:

1. Use the graphical or headless installer to install the new version of Diffusion.

   For more information, see Installing on page 45.
2. You can copy your existing license file from your previous installation to the `etc` directory of your new installation.
3. You can copy your existing configuration files from the `etc` directory of your previous installation to the `etc` directory of your new installation. When you do, consider making the following changes:

   - In the `WebServer.xml` configuration file for your production installation, remove or comment out the configuration for the HTTP deploy service.

     Access to this service is not restricted. If you enable the deploy service, you must restrict access to the deploy URL by other methods to prevent unauthorized or malicious access. For example, by setting up restrictions in your firewall.
   - Remove the `writer-selector` configuration from the `Server.xml` configuration file. Writer selectors are no longer used.

     Warning: Do not confuse writer selectors with write selectors.

- If you now use authentication handlers for authentication, configure these handlers in the `Server.xml` configuration file.
- If you now use the replication high availability features, configure these in the `Replication.xml` configuration file.
- If you use the WhoIs service, but do not explicitly configure it, you must now configure the service in the `Server.xml` configuration file.

  In previous releases, if no configuration was specified for the WhoIs service, the service started with the default configuration. In this release, the service does not start unless configuration is present in the `Server.xml` configuration file.

The validation of the configuration files has been relaxed. The order of the element within the files is less strict.

4. If you start the Diffusion server from your own scripts or Java programs, you must update them to take into account the following changes:

   - The Java license agent has been removed. Remove the following argument from the Java command you use to start the server:

     ```
     -javaagent:../lib/licenceagent.jar=../etc/licence.lic,../etc/
     publicKeys.store
     ```

   - New system properties are required by the Diffusion server.

     Include the following properties in the Java command that starts the server:

     ```
     -Ddiffusion.license.file=diffusion_installation/etc/licence.lic
     -Ddiffusion.keystore.file=diffusion_installation/etc/publicKeys.store
     -Ddiffusion.home=diffusion_installation/lib
     ```

     You can also supply these properties as VM arguments.

     For more information, see Running from within a Java application on page 58

   If you use the start scripts provided with the Diffusion installation, you do not need to make any changes.

Upgrading from version 5.0 to version 5.1

Consider the following information when upgrading from Diffusion version 5.0 to version 5.1.

Upgrading your applications

Server-side components

Recompile all Java application components that are deployed to the Diffusion server, such as publishers and authorization handlers, against the new version `diffusion.jar` file. This file is located in the `lib` directory of your new Diffusion server installation.

Some features that your Java application components might use have been removed or deprecated. Review the API changes information in the following section to see if these changes affect your applications.

Event publishers

The event publisher APIs are deprecated. Reimplement your event publisher as a control client using the Unified API control features. For more information, see Control client on page 226 and Unified API on page 240.

Clients

You can choose not to recompile your client applications and continue to use client libraries from a previous release. If you choose to use client libraries from a previous release, ensure that the libraries are compatible with the new server. For more information, see Interoperability on page 25.

You can choose to upgrade your client applications to use the new client libraries. To do this, recompile the client applications against the client libraries located in the `clients` directory of your new Diffusion server installation. Some features that your client applications might use have been removed or deprecated. Review the API changes information in the following section to see if these changes affect your applications.

API changes

Further information about removed or deprecated features is available in following locations:

- The release notes provided in the `docs` directory of your Diffusion installation
- The API documentation located at http://docs.pushtechnology.com/5.1

The following table lists API classes and methods that have been removed. If you attempt to recompile application code that uses these classes or methods against the version 5.1 APIs, it fails. Rewrite your application code to not include these features.

Table 3: API features removed in version 5.1

API affected	Removed feature	Suggested alternative
Publisher API Event Publisher API	The capability to set the maximum queue size to -1, which specified an unbounded queue size, using `Client.setMaximumQueueSize()`.	Set the maximum queue size value to a positive integer.

The following table lists API classes and methods that have been deprecated. If your application code uses these classes or methods, consider rewriting your application code to not include these features.

Table 4: API features deprecated in version 5.1

API affected	Deprecated feature	Suggested alternative

Unified API	`TopicUpdateControl.TopicSource`
	`TopicUpdateControl.TopicSource.Default`
	`TopicUpdateControl.TopicSource.Updater`
		`TopicUpdateControl.UpdateSource`
		`TopicUpdateControl.UpdateSource.Default`
		`TopicUpdateControl.Updater`

Java Unified API	`Messaging.Listener` and associated methods	`Messaging.MessageStream`
Java Unified API	`Topics.Listener` and associated methods	`Topics.TopicStream`

API affected	Deprecated feature	Suggested alternative
Java Unified API	`Updater.update()` methods that take both `Content` and `UpdateOptions` as parameters	`Updater.update()` methods that take `Update` as a parameter
Java Unified API	`comparator()` and `duplicatesPolicy()` methods in the `PagedTopicDetails.Builder` class	`order(String)`, `order(Duplicates, String)`, or `unordered()`
Java Unified API	`getComparator()` and `getDuplicatesPolicy()` methods in the `PagedTopicDetails.Attributes` class	`getOrderingPolicy()`
.NET Unified API	All `Set`*`Property`*`()` methods in the `ISessionFactory` class, where *Property* is the name of the value you want to change	*`Property`*`()`
Unified API	The `autoSubscribe` method in the `TopicDetails.Builder` interface. In future, auto-subscribe is always true.	None
JavaScript Classic API	The functions `setCrypted()` and `getCrytped()`	`setEncrypted()` and `isEncrypted()`
Java Server API	The functions `getStartTimeMillis()`, `getUptime()`, and `getUptimeMillis()` on the `c.p.d.api.topic.Subscription` class	You can use a publisher to get equivalent functionality.
Java Unified API	All static fields in the `c.p.d.client.types.Constants` class	These fields are now available in the `c.p.d.client.content.Record` class.
Java Unified API, .NET Unified API	`RecordContentReader.hasMore()`	`RecordContentReader.hasMoreRecords()`, `RecordContentReader.hasMoreFields()`

The following list includes behavior that has changed in the API. If your application code relies on the previous behavior, rewrite your application code to take into account the new behavior.

- The UpdateSource API that replaces the TopicSource API in the Unified API behaves in a very similar way, but has some differences.
  - UpdateSource includes an `onRegister` callback, which provides a `RegisteredHandler` that the client can use to deregister as an update source. Previously, TopicSource only provided the ability to deregister as an update source to the client that was the active update source.
  - UpdateSource includes an `onDiscard` callback, which indicates to a client when an update source is prematurely closed.
  - Updaters include an `onDiscard` callback, which indicates to a client when an updater is prematurely closed.
  - Updaters do not accept an `UpdateOptions` object as a parameter. Instead an `Update` object is used to contain the update content and any additional information about the update.

  For more information about how update sources work, see Topic updates from a control client.

- In the Unified API, the logging level at which the `Topics.Listener.Default` logs updates has changed from "warn" to "debug".
- In the Unified API, topic and messaging listeners are called in the order that they were registered. In previous releases, these listeners might have been called in any order.
- In the Unified API, you must set a fallback topic or messaging listener explicitly. In previous releases, a fallback topic or messaging listener was set by default.
- In the Unified API, a notification occurs for any type of selector that does not match with any topics. In previous releases, a missing topic notification occurred only if a topic path selector was used for subscribe or fetch and there was no such topic.
- In the Unified API, you can add multiple session listeners and remove session listeners. In previous releases, you could add only one session listener.

Upgrading your server installation

> **Note:** At release 5.1, the Diffusion server is tested and supported on Java HotSpot™ Development Kit 7 (update 67). If you are using an earlier update of Java 7, we recommend that you update to update 67. This update includes a number of security improvements over the previously supported version of Java 7.

To upgrade your Diffusion server installation, complete the following steps:

1. Use the graphical or headless installer to install the new version of Diffusion.

   For more information, see Installing on page 45.
2. You can copy your existing license file from your previous installation to the `etc` directory of your new installation.
3. You can copy your existing configuration files from the `etc` directory of your previous installation to the `etc` directory of your new installation. When you do, consider making the following changes:

   - In the `Server.xml` and `Connectors.xml` configuration files, if you have set maximum client queue depths to be unbounded (0), change these values. Unbounded outbound client queues are no longer allowed.

Behavior changes at the Diffusion server

The following list includes behavior that has changed at the server. If your solution relies on the previous behavior, adjust your solution to take into account the new behavior.

- In previous releases, messages that required acknowledgment were prioritized over other messages. This might have caused ordering problems. From 5.1, messages that require acknowledgment are queued for sending in the order of receipt. You might have to increase your acknowledgment timeout value to allow for the additional queuing time.
- The non-configurable 5s timeout between HTTP polls has been removed. Use `<system-ping-frequency>` for HTTP connectors.

Upgrading to a new patch release

When upgrading to a new patch release there are typically no changes to the configuration values or the APIs. All that is required is to copy your existing files from the old installation to the new installation.

To upgrade to a new patch release, complete the following steps:

1. Use the graphical or headless installer to install the new version of Diffusion.

   For more information, see Installing on page 45.
2. Copy your existing license file from your previous installation to the `etc` directory of your new installation.
3. Copy your existing configuration files from the `etc` directory of your previous installation to the `etc` directory of your new installation.

4. Copy any publishers located in the `ext` directory of the previous installation into the `ext` directory of the new installation.

Chapter 2

Overview

In this section:

- Architecture
- Data distribution
- Diffusion server
- Publishers
- Topics
- Clients
- Control clients
- Diffusion APIs

Diffusion is a component-based and modular data distribution platform.

Some Diffusion components are optional. Only the Diffusion server, publishers, and clients are required to create a working platform. One of the benefits of component-based architecture is the ability to build your architecture as your business requirements grow.

This section provides an overview of the components and functionality of the Diffusion platform.

Architecture

Diffusion is a data distribution platform that uses a low latency messaging server to merge and distribute data from multiple sources to client applications.

The following diagram shows the components of a Diffusion architecture and how those components interact.

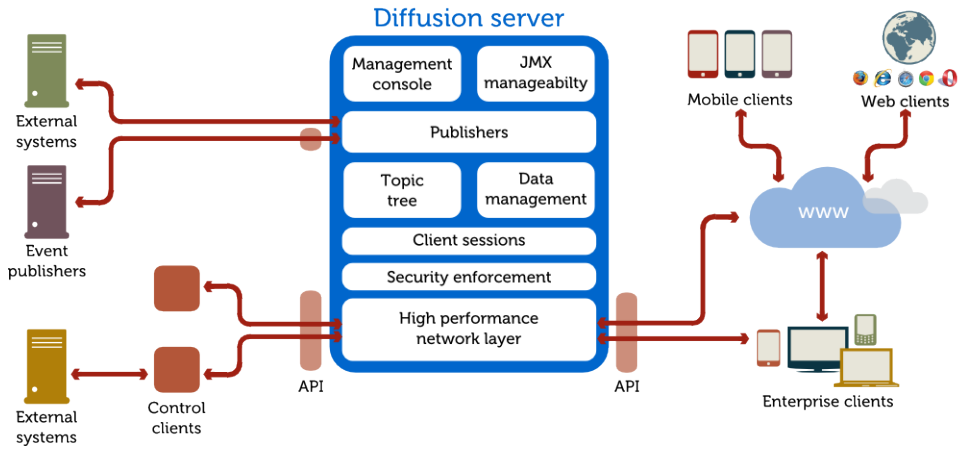


Figure 1: Basic architectural components

The following components can make up a basic Diffusion architecture:

Diffusion server

You must have at least one Diffusion server in your architecture.

The Diffusion server hosts publishers and the topic tree. It manages connections from clients and pushes data to the clients through message queues.

Publisher

Publishers create and maintain topics. They coordinate the distribution of content on topics. Publishers are written in Java and deployed within a Diffusion server.

Client

Client applications connect to the server and subscribe to topics. They receive the messages that are published to these topics from the server.

The diagram shows different categories of client: web, mobile, and enterprise. The category of client depends on the language of the API and libraries used to implement it. Clients can be implemented in one of a number of languages and use variety of protocols to communicate with the server.

Control client

A control client is a client that connects to the server and can take responsibility for control functions, for example authenticating clients or creating and updating topics.

Control clients can be implemented in one of a number of languages and use a variety of protocols to communicate with the server.

The Diffusion APIs

The Diffusion APIs and associated client libraries come packaged as part of the server. They are provided for a number of different platforms. Clients and control clients must implement the API and include the libraries appropriate to their platform to interact with the server.

Event publisher (deprecated)

An event publisher can send topic messages to the Diffusion server. The server routes these messages to the publisher that manages the topic. Event publishers can be deployed on systems remote to the Diffusion server.

The event publisher APIs are deprecated as of Diffusion 5.1. You can create a control client that has the capabilities of an event publisher.

Example of a scalable, resilient architecture

Your Diffusion architecture can use the following features to be scalable and resilient:

- Load can be spread across different publishers on different servers.
- Publishers can act as clients by subscribing to topics on other servers.
- Servers can share information with each other by replicating it into a datagrid.
- Servers can load balance requests across multiple control clients that have registered to handle the request.

The following diagram shows how these features can be used in an architecture.

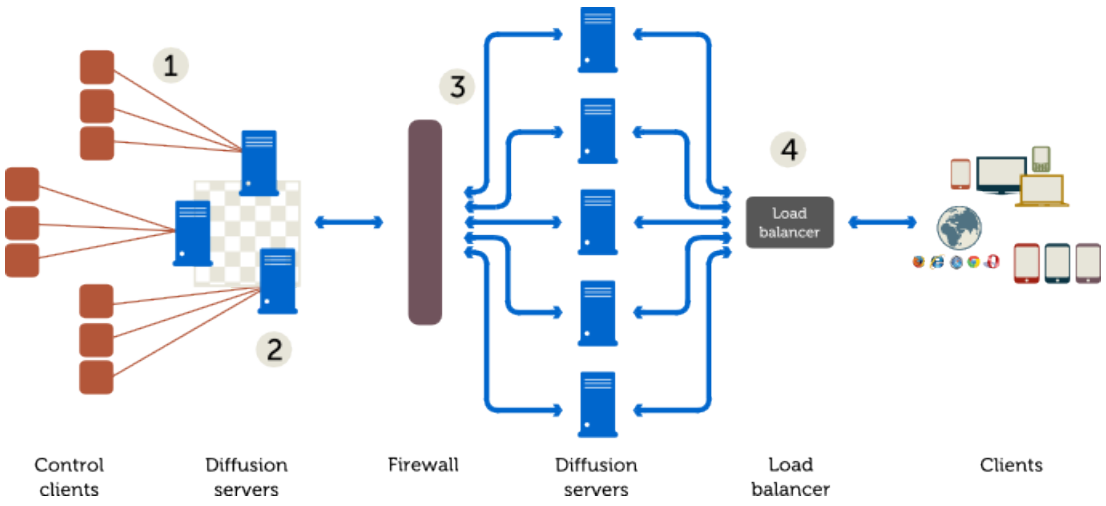


Figure 2: A scalable, resilient architecture

1. Three control clients register handlers with each of the Diffusion servers behind the firewall. These control clients can be located on the same system as the server or on remote systems. The server load balances requests between control clients that have registered to handle requests of that type. If one of the control clients becomes unavailable, the requests can be directed to another control client. You can connect more control client sessions to deal with higher volumes of requests.
2. The Diffusion servers inside the firewall replicate information into a datagrid. If a server that was handling a client session or topic becomes unavailable, the responsibility for that client session or topic can be passed to another server that has access to all the information for that session or topic through the datagrid.
3. Publishers on the Diffusion servers outside of the firewall, in the DMZ, can subscribe as clients to the topics on the Diffusion servers inside the firewall.
4. You can use a load balancer to spread requests from clients across many Diffusion servers. If a server becomes unavailable, clients can be directed to another server.

Related Links

Diffusion server on page 38
The Diffusion server is a highly efficient, low latency messaging server that can be deployed as a standalone server or as part of a cluster to provide a fully scalable enterprise data distribution solution.

Publishers on page 39
Publishers are components hosted within a Diffusion server that manage the data for one or more topics and publish messages to any clients that subscribe to the topics that the publisher manages.

Clients on page 41
A client is any application that communicates with a Diffusion server using a Diffusion client protocol.

Control clients on page 42
Control clients are client applications that can perform control functions on the server and to handle events that occur on the server.

Diffusion APIs on page 42
Diffusion provides application programming interfaces (APIs) that you can use to create applications that interact with the Diffusion server.

Data distribution

Diffusion provides a high performance infrastructure for distributing data. There are two mechanisms by which Diffusion distributes data: by publishing messages to topics or by sending messages between applications.

What is a message?

Data sent through Diffusion is formatted in messages. A message is a contiguous sequence of bytes comprising some header information followed by a data payload.

There is no limit on the format of data held within a message. Messages can be encoded to either compress or encrypt the data content of the message.

Messages are sent between server, clients and other components over socket-based connections using the Diffusion message protocol.

Publishing to topics

The primary mechanism of distributing data through Diffusion is by using topics to provide pub-sub messaging.

Publishers, event publishers, and control clients can create topics on the Diffusion server and publish messages to topics.

Clients can subscribe to topics and receive messages that are published to that topic.

You can use this mechanism to send from many sources to many recipients.

Sending messages

Data can be distributed between endpoints by sending messages through Diffusion.

Publishers, event publishers, and control clients can send messages to clients or groups of clients.

Clients and event publishers can send messages to a topic. The Diffusion server routes those messages to the publisher that owns the topic or the control client that is registered to handle messages sent on that topic.

Related Links

Messages on page 193

Publishers publish messages on topics which clients subscribe to so they can receive those messages. Clients can also send messages to publishers.

Diffusion server

The Diffusion server is a highly efficient, low latency messaging server that can be deployed as a standalone server or as part of a cluster to provide a fully scalable enterprise data distribution solution.

The Diffusion server provides the following main functions:

- A scalable and mature solution to push (stream) and receive data and events, in real-time, both to and from client applications.
- High performance transport of data between two or more application within your own network or extranet.

The Diffusion server has been certified to run on the Oracle® Java HotSpot Development Kit 7 (update 67).

Figure 3: Components in a Diffusion server

High performance network layer

The high performance network layer uses the Java NIO framework to handle a high number of concurrent connections without the need for separate threads. Connectors handle connections from many different types of client and on many protocols. You can configure many connectors to listen on different ports. Multiple clients can connect to a single port.

Security enforcement

The Diffusion server authenticates all connections from clients. It also manages authorization and permissioning for actions that those clients can take when they are connected to the server.

Client sessions

The server manages the sessions for all of the clients that connect to it. It stores information about the client and the client's subscriptions. If a client disconnects, it can reconnect to the same session within a certain time period.

Topic tree

Topics are arranged in a tree structure. Inside the topic tree are top-level topics with subordinate topics. These subordinate topics can themselves

have subordinate topics. The topic tree is a model of the data that is published to clients.

Data management

The Diffusion server performs operations on the data to more efficiently deliver it to clients. It performs structural conflation, merging and replacing data to ensure that the latest data is received by the client. It fragments low priority data and interleaves it with high priority data to maximize bandwidth utilization.

Publishers

Publishers are user-written applications that are hosted and run within the Diffusion server. Publishers are written in Java. They create and maintain topics.

Management console

Diffusion provides console as an optional publisher that is deployed by default. You can use it to monitor the operations of your Diffusion server through a web browser and to stop and start publishers within your server.

JMX manageability

Diffusion registers MBeans for many of its principal features with the JMX service. You can use a JMX management console, such as JConsole, to manage the Diffusion server.

Related Links

Architecture on page 35
Diffusion is a data distribution platform that uses a low latency messaging server to merge and distribute data from multiple sources to client applications.

Server on page 56
The Diffusion server is the core component of the Diffusion Product.

Publishers

Publishers are components hosted within a Diffusion server that manage the data for one or more topics and publish messages to any clients that subscribe to the topics that the publisher manages.

Publishers are written using the Java API and must extend the issued `Publisher` class and implement various methods to provide the publisher functionality

A publisher must maintain its own data model. The publisher can initialize its data as it starts and update it as a result of external events.

When a client first subscribes to a topic the publisher can provide the client with a snapshot of the current state of the data relating to that topic. This is called a topic load.

A client can also request the current state of a topic, even if not subscribed to it, using the fetch command. There is even the possibility of a publisher responding to a fetch request of a topic that does not exist. This provides a potential request/response mechanism without the overhead of real topics.

A publisher must maintain any changes to its topic data state and publish those changes to the topic as delta messages. This results in the message being sent to every client that is subscribed to the topic.

Publishers can send messages to individual clients or to groups of clients and can receive messages from clients.

For normal publishing the publisher does not need to know or keep track of the clients subscribed to its topics.

Publishers own the topics they create. Ownership of a topic is used to determine which publisher receives a message from a client, deals with subscription, and or creates dynamic topics.

Related Links

Architecture on page 35
Diffusion is a data distribution platform that uses a low latency messaging server to merge and distribute data from multiple sources to client applications.

Publishers on page 78
A publisher lies at the core of the Diffusion infrastructure. A publisher publishes messages to clients that are subscribed to the topics provided by the publisher.

Topics

Clients and publishers or control clients are loosely coupled through logical links called topics.

A publisher or control client publishes messages to a topic and a client subscribes to a topic and receives its messages.

A topic can also be used by a client to send messages to the publisher or control client that receives messages on that topic. The client is not aware of the publisher or control client, only of the topic.

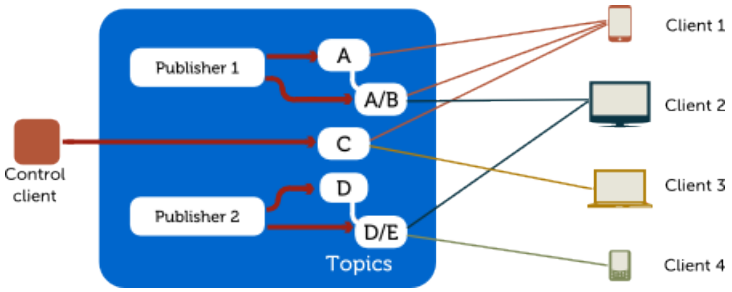


Figure 4: Topics

Topics are created in a Diffusion server by publishers or control clients. Each topic must have a unique name within the server. In the diagram the topic names are A, B, C, D, and E.

Topics can be arranged in a tree structure. In the diagram topic B is beneath topic A in the topic tree and topic E is beneath topic D in the topic tree.

The location of the topic in the topic tree is described by the topic path. The topic path includes all the topics above it in the topic tree in order separated by the slash character (/). In the diagram the path to topic B is A/B and the path to topic E is D/E.

The topic tree can include any number of topics. There are no limits to the number of topics a control client or publisher can update. There are no limits to the number of topics a client can subscribe to.

Related Links

Topics on page 126

Diffusion publishes messages to topics, to which clients can subscribe. All clients subscribed to a topic receive all messages published to the topic. The topic is a fundamental part of Diffusion.

Clients

A client is any application that communicates with a Diffusion server using a Diffusion client protocol.

Most clients connect to the Diffusion server only to subscribe to topics and receive message data on those topics. Some clients, known as control clients, perform control actions such as creating and updating topics or handling events. For more information, about control clients, see Control clients on page 42.

Some of the types of client that are supported by Diffusion are listed in the following sections.

Web clients

Clients can be browser applications using one of the following APIs provided by Diffusion:

- JavaScript
- ActionScript
- Silverlight®

Enterprise clients

A client can be any application connecting to Diffusion over a DPT socket connection which can be over the internet or an intranet/extranet. Enterprise clients can use one of the following APIs provided by Diffusion:

- Java
- .NET

Applications in other languages can communicate using the raw DPT.

Mobile clients

Clients can be mobile applications using one of the following APIs provided by Diffusion:

- iOS
- Android
- Windows™ phone

Publisher clients

Publishers hosted in a Diffusion server can act as client applications to other Diffusion servers. A publisher can do this by subscribing to topics on the other server. This type of communication enables you to create a distributed Diffusion architecture.

Related Links

Architecture on page 35
Diffusion is a data distribution platform that uses a low latency messaging server to merge and distribute data from multiple sources to client applications.

Clients on page 104
In Diffusion terms a client is any application that connects to a Diffusion server using a Diffusion client API.

Control clients on page 42
Control clients are client applications that can perform control functions on the server and to handle events that occur on the server.

Diffusion APIs on page 42

Diffusion provides application programming interfaces (APIs) that you can use to create applications that interact with the Diffusion server.

Control clients

Control clients are client applications that can perform control functions on the server and to handle events that occur on the server.

Control clients can act like a publisher by creating and updating topics. Unlike a publisher, a control client does not have to be located inside a Diffusion server instance and can be written in one of a number of languages.

Control clients can also handle events that occur at the server or on topics. For example, a control client can register to handle the following events:

- A client attempts to connect to a server. The control client can authenticate the client.
- A client sends a message to a topic. The control client can receive messages from a topic or section of the topic tree.

There are other control capabilities that a control client can have. Control clients are implemented using the Unified API.

Related Links

Architecture on page 35
Diffusion is a data distribution platform that uses a low latency messaging server to merge and distribute data from multiple sources to client applications.

Control client on page 226
A Diffusion client application can take on the role of control client.

Diffusion APIs

Diffusion provides application programming interfaces (APIs) that you can use to create applications that interact with the Diffusion server.

The Unified API

You can use the Unified API to write clients and control clients. The Unified API is a modular, feature-based API and is available for Java, .NET, and C.

The Classic API

The Classic API comprises a number of APIs.

Client APIs

You can use the client APIs to create clients that subscribe to topics and receive messages from topics. The client API is available for the following platforms: Java, .NET, JavaScript, ActionScript, Silverlight, iOS, Android, and C.

Event publisher API

You can use the event publisher API to write an event publisher that publishes messages to topics. The event publisher API is available for Java and .NET.

Publisher API

You can use the publisher API to create a publisher that runs inside the Diffusion server. A publisher must extend the `Publisher` class. THe publisher API is available in Java.

Related Links

Architecture on page 35
Diffusion is a data distribution platform that uses a low latency messaging server to merge and distribute data from multiple sources to client applications.

Classic APIs on page 342
Diffusion provides a number of Application Programming Interfaces (APIs) which allow user-written applications to make use of Diffusion.

Unified API on page 240
The Diffusion Unified application programming interface (API) provides a consistent interface to the Diffusion server, whichever role your client application performs: standard client or control client.

Part

II

Diffusion Guide

In this section:

- Installing
- Server
- Web server
- Publishers
- Clients
- Topics
- Message conflation
- Messages
- Event publishers (deprecated)
- Control client
- Unified API
- Classic APIs
- System management
- Security
- Distribution
- Configuration
- Adapters
- Tuning
- Diagnostics
- Introspector
- Demos
- Testing
- Tools
- Appendix: Protocol

Chapter 1

Installing

In this section:

- System requirements for the Diffusion server
- Obtaining a Diffusion license
- Installing the Diffusion server
- Installing Diffusion using the headless installer
- Installing Diffusion using Red Hat Package Manager
- Updating your license file
- Installed files
- Verifying your installation
- Web server installation

You can install the Diffusion server from a JAR file or through Red Hat Package Manager.

Review the system requirements before installing Diffusion.

Download Diffusion from the following location: http://download.pushtechnology.com/releases/5.1

The Diffusion installation includes a developer license that allows up to five concurrent connections to the Diffusion server. To use Diffusion in production, you can obtain a production license from Sales at Push Technology.

System requirements for the Diffusion server

The Diffusion server is certified on a range of hardware and operating systems. Review this information before installing the Diffusion server.

We recommend that you use certified hardware, virtual machines, operating systems, and other software when setting up your Diffusion servers.

However, Push Technology can agree to support Diffusion on other systems. For more information, contact Push Technology.

Hardware

The Diffusion server is tested and certified on the hardware with the following specifications:

- Intel™ Xeon™ E-Series Processors
- 8Gb RAM
- Intel Gigabit NICs with TCP Offloading and Intel/Solar Flare 10GBE cards for performance testing.

NIC, CPU, and RAM (in decreasing order of importance) are the components that have the biggest impact on performance. A rapid file system is not considered a necessity although Diffusion server is tested using 15K SAS drives and RAID 10. Intel hardware is used due to its ubiquity in the marketplace and proven reliability.

Virtual machines

The Diffusion server is tested and certified on VMware® with the following host specifications and virtual machine specifications.

Hardware specifications:

- 2 Dell™ E5-2650
- 32Gb RAM
- 240GB OCZ Agility 3 SSD
- VMware vSphere® 5 Dell release

Virtual machine specifications:

- 8 VCPUs
- 28Gb RAM
- 30GB SAS emulated HDD
- Intel X540 (Non-direct passthrough)
- CentOS™ 6.5

Operating system

Diffusion is tested and certified on the following operating systems:

- Red Hat 6.5
- CentOS 6.5
- Windows Server 2008 R2

We recommend you install your Diffusion server on a Linux™-based operating system with enterprise-level support available, such as Red Hat or SUSE® Enterprise. Diffusion on Linux provides enhanced performance.

Java

The Diffusion server is tested and certified on 64-bit Oracle Java 7 distributions. We recommend you use Java HotSpot Development Kit 7 (update 67).

Open source equivalents are not supported. We suggest you use the package manager of your operating system to add the official JDK.

Networking

The use of F5® load balancers with SSL offloading is recommended.

Client requirements

For information about the supported client platforms, see Supported client platforms on page 258 and Interfaces supported on page 116.

Obtaining a Diffusion license

Diffusion includes a development license that enables you to use make up to 5 concurrent connections to the Diffusion server.

To use Diffusion in production, contact Sales at Push Technology for production licenses.

You can install a license file into an existing Diffusion server installation by placing the license file in the *diffusion_installation*`/etc` directory and restarting the server.

Installing the Diffusion server

The Diffusion binary files are available from the Push Technology website. You can install Diffusion using the graphical installer.

Before you begin

You must have Java installed on your system to install and use Diffusion.

About this task

To install Diffusion using the graphical installer, complete the following steps:

Procedure

1. Go to the Diffusion download page:
   http://download.pushtechnology.com/releases/5.1
2. Click on the following download links to download the required jar files into a temporary directory:
   - Diffusion (`Diffusion` *version_id*`.jar`)
   - Installer (`install.jar`)
3. In the temporary directory, double-click the `install.jar` file.
   The graphical installer launches.
4. Optional: If you have a production license, you can load it into the Diffusion installation at this point.
   You can skip this step if you are using the included development license.
   a) Ensure that the license file is available on your system.
   b) At the **Introduction** step, select **File** > **Load license file**
   c) In the window that opens, navigate to the license file (`licence.lic`). Click **Open**.
5. At the **Introduction** step, click **Continue**.
6. At the **License agreement** step, select **Accept** to accept the End User License Agreement (EULA) and click **Continue**.
7. At the **Destination directory** step, select the install destination.
   We recommend you create a `Diffusion` directory on your system. Click **Continue**.

8. At the **Select products** step, select the components you want to install.
   We recommend you select **All**. Click **Continue**.
9. At the **Confirmation** step, review the install information. If the information is correct, click **Continue** to confirm.
   The installer installs Diffusion into the directory specified.
10. At the **Summary** step, click **Done** to exit the graphical installer.

Results
You have successfully downloaded and installed Diffusion.

Related Links
Installed files on page 50
After installing Diffusion the following directory structure exists:

Installing Diffusion using the headless installer

The Diffusion binary files are available from the Push Technology website. You can install Diffusion from the command line.

Before you begin
You must have Java installed on your system to install and use Diffusion.

About this task
You can install in headless mode in circumstances where the graphical installer cannot be used or is not appropriate.

Procedure

1. Go to the Diffusion download page:
   http://download.pushtechnology.com/releases/5.1
2. Click on the following download links to download the required jar files into a temporary directory:
   - Diffusion (`Diffusion version_id.jar`)
   - Installer (`install.jar`)
3. Copy these files to a temporary directory on the system where Diffusion is to be installed.
4. In the terminal window, change to the directory where the Diffusion jar files are located.
5. Type the following command:

   ```
   java -jar install.jar Diffusionn.n.n.jar
   ```

   where *n.n.n* is the Diffusion release number.
6. If you agree to the terms of the license agreement, type `y` and Enter.
7. Enter the full path to the directory in which to install Diffusion and type Enter.
8. Type `Y` to install all packages.
   If you choose not to install all packages, the installer asks you about each package individually.

Results
Diffusion is installed in the specified directory.

What to do next

Your Diffusion installation includes a development license that allows connections from up to five clients. To use Diffusion in production, you can obtain a production license from Sales at Push Technology.

Copy the license file into the `/etc` directory of your Diffusion installation and restart Diffusion.

Related Links

Installed files on page 50
After installing Diffusion the following directory structure exists:

Installing Diffusion using Red Hat Package Manager

Diffusion is available as an RPM file from the Push Technology website.

About this task

On Linux systems that have Red Hat Package Manager installed, you can use it to install Diffusion.

Procedure

1. Go to the Diffusion download page:
   http://download.pushtechnology.com/releases/5.1
2. Click on the following download link to download the required RPM file:
   - Diffusion RPM (`Diffusion_n.n.n.rpm`)
3. Copy this file to a temporary directory on the system where Diffusion is to be installed.
4. In the terminal window, change to the directory where the Diffusion RPM file is located.
5. Type the following command:

   ```
   rpm -ivh  Diffusion_n.n.n.rpm
   ```

   where *n.n.n* is the Diffusion release number.

Results

Diffusion is installed in the following directory: `/opt/Diffusion`. A startup script is installed in the `/etc/init.d` directory that enables Diffusion to start when you start the system.

What to do next

Your Diffusion installation includes a development license that allows connections from up to five clients. To use Diffusion in production, you can obtain a production license from Sales at Push Technology.

Copy the license file into the `/etc` directory of your Diffusion installation and restart Diffusion.

Related Links

Installed files on page 50
After installing Diffusion the following directory structure exists:

Updating your license file

You can update your Diffusion license file without having to restart the Diffusion server. Copy the new file over the old and ensure that the timestamp is updated.

Before you begin

Obtain a new or renewed license file from Push Technology.

About this task

When your license file expires, the Diffusion server continues to run for another day before it stops. We recommend you update your license file before your existing license file expires.

Procedure

1. Copy the new license file (`licence.lic`) over the existing file in the *`diffusion_directory`*`/etc` directory.

   You do not need to stop or restart the server.
2. Check that the timestamp of `licence.lic` has updated.
   - On Windows, you might have to use the following command to copy the file over and force the timestamp to update: `COPY /B licence.lic +,,`
3. Diffusion checks the timestamp of the `licence.lic` every minute. If the license file has been updated, Diffusion reloads it and logs this to stdout.
4. You can verify that the license file has been updated in the server by accessing the mbean **com.pushtechnology.diffusion** > **Server** > **LicenseExpiryDate**

Installed files

After installing Diffusion the following directory structure exists:

Table 5: Installed files

Folder name	Contents
`bin`	Executables for starting Diffusion
`clients`	Client Diffusion API libraries and related artifacts for all supported platforms.
`data`	Files used by issued demo publishers. This directory is always on the server classpath. You can place files that must be loaded at runtime in this directory.
`demos`	The compiled DAR files for the demos issued with Diffusion an buildable source code. For more information, see Demos on page 572.
`deploy`	Publisher DAR files that are deployed when the Diffusion server starts. If you selected during the install process to deploy the demos, the demo DAR files are in this directory.
`docs`	License information, release notes, and install notes.
`etc`	The properties files for configuring the Diffusion server and example policy files for Silverlight and Flash®. For more information, see Configuration on page 441.
`examples`	Example code that uses the Diffusion APIs.
`ext`	External jar files that Diffusion runs.
`html`	Files that are used by the default web server for issuables accessible through the browser.
`lib`	The main Diffusion server JAR file, third-party libraries, and additional server-side components.
`logs`	The directory to which Diffusion server and web server logs are written.

Folder name	Contents
`stresstest`	The stress test package. For more information, see Stress test tuning on page 593.
`tools`	Tools and utilities that help with testing and deploying Diffusion. For more information, see Tools and utilities on page 51
`xsd`	The schema files for the XML configuration files used by the server.

Tools and utilities

The following table describes the some of the contents of the `tools` directory.

Note: The files present and their suffixes vary according to the platform that the product is installed on.

Table 6: Tools and utilities

Tool	Description
`/ec2`	A sample configuration for setting up Diffusion in an Amazon™ EC2 instance.
`/init.d`	Sample `init.d` files to start Diffusion as daemon on OS X®, Linux, or UNIX® systems.
`/joyent`	A sample configuration for setting up Diffusion in a Joyent™ instance.
`DiffusionEventPublisherTest.exe`	Windows event publisher test tool.
`DiffusionExternalClientTest.exe`	Windows external client test tool.
`eventpub.bat/sh`	Generic event publisher test tool.
`extclient.bat/sh`	Generic external client test tool.
`externalclienttest.properties`	External client test tool properties.
`war.xml`	Example `war.xml` file
`web.xml`	Example `web.xml` file
`jstack.sh`	Prints stack traces for threads within a Java VM. Based on the Java jstack tool.

Verifying your installation

Verifying the Diffusion installation

About this task

After installation, all of the Diffusion files are available in the directory specified during installation.

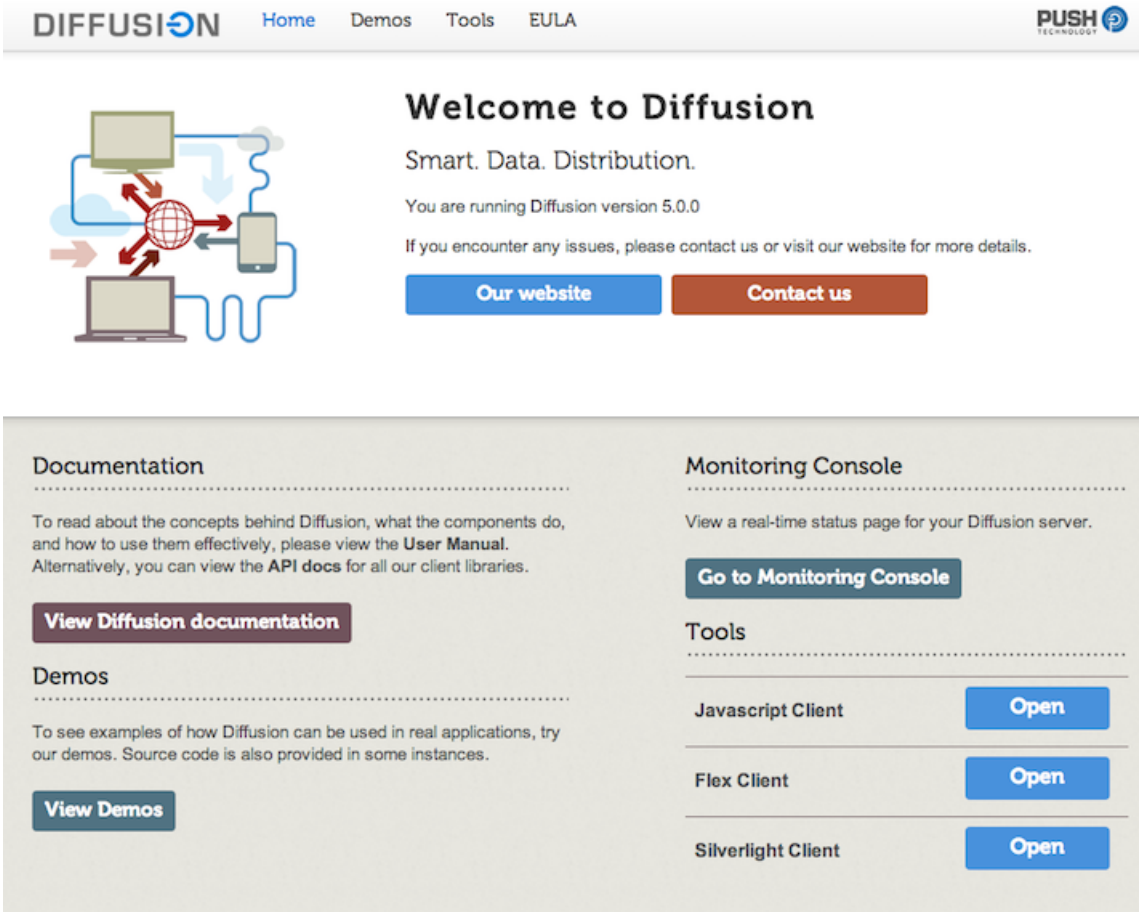
Procedure

1. Start the Diffusion server using one of the start script located in the `bin` directory of your Diffusion installation.
   - On Windows, use the `diffusion.bat` file.
   - On Linux, OS X, or UNIX, use the `diffusion.sh` file.

2. Inspect the log messages to ensure that the Diffusion server started successfully.

   The terminal window displays logging information about the status of the Diffusion server. A log message containing the following text indicates that the server started successfully: `INFO|main|PUSH0165|Diffusion Server started.|com.pushtechnology.diffusion.DiffusionController` This line is typically the last one to be printed on terminal.
3. Inspect all log messages displayed in the terminal to search for `WARN` messages to ensure that all components have started correctly.
4. Open a browser and navigate to `http://serverAddress:8080` (or `http://localhost:8080`)

   The browser shows the Diffusion landing page.


DIFFUSION
Home
Demos
Tools
EULA
PUSH
Welcome to Diffusion
Smart. Data. Distribution.
You are running Diffusion version 5.0.0
If you encounter any issues, please contact us or visit our website for more details.
Our website
Contact us
Documentation
To read about the concepts behind Diffusion, what the components do, and how to use them effectively, please view the User Manual. Alternatively, you can view the API docs for all our client libraries.
View Diffusion documentation
Demos
To see examples of how Diffusion can be used in real applications, try our demos. Source code is also provided in some instances.
View Demos
Monitoring Console
View a real-time status page for your Diffusion server.
Go to Monitoring Console
Tools
Javascript Client
Open
Flex Client
Open
Silverlight Client
Open

   The landing page provides pointers to sources of information regarding legal terms and conditions (for example, EULA), user guides, API documentation and demos.

   The Diffusion server is ready to be used.
5. If you chose to install the demos, you can access them from the landing page. Use these demo publishers to verify your installation.

Web server installation

Diffusion can act as a web server by modifying the `etc/Connectors.xml` and adding a web-server definition to a connector.

About this task

This explains how to setup a servlet for Diffusion. You might find it easier to create and deploy a web application using the Ant™ script discussed below. Add the following lines to `WEB-INF/web.xml`. The `servlet` section needs to go with the other `servlet` sections. A `servlet-mapping` entry is not required for the `DiffusionServlet` as it does not handle any requests.

```
<servlet>
    <servlet-name>Diffusion</servlet-name>
    <display-name>Diffusion Servlet</display-name>
    <servlet-class>com.pushtechnology.diffusion.servlet.DiffusionServlet</
servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
```

Make sure that the JAR files in `ext` and the `lib/diffusion.jar` are in the lib directory of `WEB-INF`. You should also copy the `lib/thirdparty` directory and its contents to `WEB-INF/lib`. Also copy all of the properties files from the `etc` directory to the `classes` directory. Make sure that the policy files are correctly pathed in the `etc/Connectors.xml` file. The `html/lib/DIFFUSION` directory must also be copied into the `webapp` directory as `lib/DIFFUSION`, this directory contains the browser API libraries.

Creating a web application archive

In the `tools` directory of the default installation is an Ant build file `war.xml` that will package the Diffusion into a WAR file that you can deploy to a servlet container. Invoking the `war` or `all` target of the file builds the archive at `build/diffusion.war`. The diffusion.war file can be deployed to your servlet container according to its documentation.

A WAR file is an archive like a JAR that contains a WEB-INF directory. This directory contains XML based configuration describing the web application and the directories `classes` and `lib` that contain `.class` or `.jar` files that are added to the classpath. The top level of the WAR file contains resources that can be served by Tomcat™. Place the configuration files for Diffusion in the `WEB-INF/classes` directory. Place the `diffusion.jar` and files from the `ext` directory and the `thirdparty` directory itself in the `WEB-INF/lib` directory.

Tomcat Configuration

Diffusion uses the `java.util.prefs` functionality so Tomcat must be started with the userRoot configured. Without this warning messages are generated in the logs. The Tomcat user requires write permissions to the userRoot directory.

```
-Djava.util.prefs.userRoot=/var/lib/tomcat6/diffusion/prefs/user
```

Tomcat must have connectors defined to handle incoming connections. A connector defines the port, protocol, and various properties of how the connection is handled. Connectors are defined in the `Server.xml` file. See the Tomcat documentation for more information. The following is an example connector for handling HTTP 1.1 connections on port 8080.

```
<Connector port="8080"
        connectionTimeout="20000"
        URIEncoding="UTF-8"
        maxThreads="3"
        protocol="HTTP/1.1" />
```

Accessing publishers from Tomcat

Diffusion started within Tomcat allows Tomcat to access the publishers. Tomcat can be used to serve JSP files providing dynamically generated content. These files can access publishers using the publishers class static methods.

```
<%@ page import="java.util.List,com.pushtechnology.api.publisher.*" %>
<html>
    <head>
        <title>Publisher Information</title>
    </head>
    <body>
        <table>
            <tr>
                <th>Publisher Name</th>
                <th>Topics</th>
            </tr>
            <% for (Publisher pub : Publishers.getPublishers()) { %>
            <tr>
                <td><%= pub.getPublisherName() %></td>
                <td><%= pub.getNumberOfTopics() %></td>
            </tr>
            <% } %>
        </table>
    </body>
</html>
```

The above is the content of a JSP file that return a list of the publisher Diffusion is running with the number of topics each publisher owns.

Apache™ Mod Proxy installation

Apache Mod Proxy can be used to forward HTTP requests from an Apache web server to Diffusion. It does not support persistent connections orWebSockets so the WebSocket and HTTPC connections do not work. Make sure that you include the following into the Apache configuration file (Virtual host setting).

```
ProxyPass /diffusion/ http://localhost:8080/diffusion/
```

For more information, see the Apache Mod Proxy documentation.

Apache AJP13 Installation

Apache AJP can be used to forward requests from an Apache web server to Tomcat. In the Apache virtual host configuration, mount the path

```
JkMount /diffusion/*dfnjetty
```

Workers definition file

```
worker.dfnjetty.port=8009
worker.dfnjetty.host=(host IP)
worker.dfnjetty.type=ajp13
worker.dfnjetty.lbfactor=1
worker.dfnjetty.cachesize=50
worker.dfnjetty.socket_keepalive=1

worker.list=dfnjetty
```

A connector that handles the AJP/1.3 protocol is needed running on port 8009 (because of the Workers file described above). See the Tomcat documentation for more information on this.

IIS Installation

Use an ISAPI_Rewrite tool. For example, http://www.helicontech.com/isapi_rewrite

The rewrite rule is as follows:

```
RewriteEngine on RewriteRule ^diffusion/ http://localhost:8080/diffusion/
 [p]
```

Diffusion home directory

The servlet container must be aware of the Diffusion. Add the path to the directory that contains the Diffusion JAR file to the Java VM arguments that you use to start the servlet container.

```
-Ddiffusion.home=diffusion_installation/lib
```

Licensing

The Diffusion license must be loaded by the servlet container. Add the path to the license file to the Java VM arguments that you use to start the servlet container.

```
-Ddiffusion.license.file=diffusion_installation/etc/licence.lic
```

The license file is usually located in the `etc` directory of the Diffusion installation. As this file must be referenced when the container starts, it is not a part of the Diffusion servlet. The path must be absolute or relative to the servlet container, not the web application.

Keystore

You must tell the servlet container which keystore to use. Add the path to the keystore to the Java VM arguments that you use to start the servlet container.

```
-Ddiffusion.keystore.file=diffusion_installation/etc/publicKeys.store
```

The keystore file is usually located in the `etc` directory of the Diffusion installation. As this file must be referenced when the container starts, it is not a part of the Diffusion servlet. The path must be absolute or relative to the servlet container, not the web application.

Diffusion configuration

The built-in Diffusion web server is configured using the `WebServer.xml` file. When using a third party web server at least some of this functionality can be disabled. The file-service, and two http-service entries can be removed as Tomcat provides this functionality. The client-service is needed to supportWebSocket, HTTPC and HTTP connection protocols. If these are not used it can be disabled as well.

Chapter 2

Server

In this section:

- Server basics
- Starting the server
- Running from within a Java application
- Concurrency
- Connectors
- Load balancers
- Replication

The Diffusion server is the core component of the Diffusion Product.

The server is written in Java and can be deployed on any platform that supports Java.

See the Installation section for details of how to install a server instance.

Server basics

What is the Diffusion server?

The Diffusion server is a highly efficient and low latency messaging server which provides the functions listed here.

- A scalable and mature solution to push (stream) and receive data and events, in real-time, both to and from a web browser and other net-connected devices.
- High performance transport of data between two or more machines within your own network or extranet.
- Web server capability.

A single server can be deployed to push messages to web browsers (or other net devices) or many interconnected servers can provide a fully scalable enterprise messaging solution.

Starting the server

Here we learn the basic of how to configure and start a Diffusion Server.

Before you begin
A Diffusion server instance can be installed as described in the Installation section. After installing, the installed files are present in the specified install directory. As issued, a Diffusion server is configured to run the demo publishers and these can be tried out and explored to familiarize with the product.

However, for a real implementation it is first necessary to design and write publisher and client applications.

Once a publisher is made available, Diffusion can be configured to run it. This is done by editing the XML property files – see the Configuration section for more details.

To configure the server to run a user written publisher involves the following:

Procedure

1. Define connectors appropriate for the types of connections (client, event publisher, and so on) that are supported using `etc/Connectors.xml`
2. Define publishers using `etc/Publishers.xml`.

   Other configuration might be required according to the nature of the application. The above steps are the bare minimum requirement to get a publisher working.

   The server produces logs which record actions performed by the server and can also be used for diagnostic purposes. The level of logging produced and where logging is directed is configurable. See Logging for further details.
3. The `diffusion.sh` or `diffusion.bat` command (issued in the `bin` directory) starts Diffusion. An optional properties directory can be specified as a parameter to be used instead of the default `../etc` directory.

   > **Important:** Do not run your Diffusion server as root on Linux or UNIX. To run the Diffusion server on a port number of 1024 or lower, use another means. For some examples of ways of doing this, see http://www.debian-administration.org/articles/386.

   Once a server is started, it must be accessible from client applications or test tools.
4. A running server can be managed using JMX or using the management API
5. Tuning the server is critical to getting the best performance out of a Diffusion installation.

   Some of the aspects that must be considered are message design and sizing, Concurrency, client queues and connection buffers. All of these issues are discussed in detail in the Tuning section.

Running from within a Java application

To run Diffusion from within a Java application instantiate, configure and start a `DiffusionServer` object.

Creating a server

`DiffusionServer` is available in the `com.pushtechnology.diffusion.api.server`. You can instantiate it with one of the following constructors:

No configuration

```
DiffusionServer server = new DiffusionServer();
```

This instantiates the server with default configuration options. Required aspects of the server must be configured programmatically before it is started.

XML configuration

```
DiffusionServer server = new DiffusionServer(directoryPath, true);
```

This specifies the path to the directory to load the XML configuration files from as a String. XML files in the specified directory are loaded into configuration objects that can form a basis for additional programmatic configuration. A full set of files can be present and tuned as required or just a partial set of the files can be present and all missing configuration supplied programmatically.

Bootstrap properties

```
DiffusionServer server = new DiffusionServer(bootstrapProperties);
```

This specifies the path to the directory to load the XML configuration files from inside a `Properties` object. In addition to the configuration directory, you can use the bootstrap properties to define the license file, the Diffusion home directory, and the keystore file

Configuring the server

Once the server object has been instantiated it can be configured. The root configuration object can be obtained from the server object as follows:

```
ServerConfig config = server.getConfig();
```

Alternatively the root can be obtained using `ConfigManager,getServerConfig()`.

Most configuration properties have sensible defaults but some configuration objects must be supplied for a server to function. For example:

Publishers

Unless all publishers are to be hot deployed at least one publisher must be defined.

Connectors

At least one connector must be defined for clients to connect to. If no connector is defined, at startup a default connector is created and a warning logged.

Queues

Definitions of client queues must be provided. If none are supplied, defaults are set up on startup and a warning logged.

Multiplexers

At least one multiplexer definition must be configured. If none are supplied, a default one is created at startup and a warning logged.

Thread pools

Thread pool definitions are required to specify the characteristics of the thread pools the server uses. If none are supplied, a default definition is created on startup and a warning is logged.

So a typical minimum configuration set up might be:

```
DiffusionServer server = new DiffusionServer();

ServerConfig config = server.getConfig();

// Publisher
PublisherConfig publisher = config.addPublisher("My
 Publisher","com.company.MyPublisherClass");

// Connector ConnectorConfig connector = config.addConnector("Client
 Connector");
// Configure connector as required....

// Thread Pools
ThreadsConfig threads = config.getThreads();
ThreadPoolConfig inbound = threads.addPool("Inbound");
inbound.setCoreSize(3);
inbound.setMaximumSize(10);
inbound.setQueueSize(2000);
inbound.setPriority(8);
threads.setInboundPool(inbound.getName());
threads.setBackgroundPoolSize(2);

// Queues QueuesConfig queues = config.getQueues();
QueueConfig queue = queues.addQueue("DefaultQueue");
queue.setMaximumDepth(10000);
queues.setDefaultQueue("DefaultQueue");

// Multiplexer MultiplexerConfig multiplexer =
 config.addMultiplexer("Multiplexer");
multiplexer.setSize(4);
```

Starting the server

After the server configuration has been completed, the server can be started using `server.start()`.

The declared publishers are then loaded and connectors start to listen on the configured ports.

Stopping the server

The server can be stopped using `server.stop()` at which point the server is no longer available.

Run requirements

The Diffusion server application must be run from within the `bin` folder in the Diffusion install folder (or a folder at the same level) for all of the relative paths in the configuration to work properly and for Diffusion to be able to locate the runtime jars it requires. The application can be run from elsewhere but this involves changing a number of configuration items that specify relative paths and also adding jars from the `ext` directory to the application classpath.

To run a Diffusion server the following items from within the Diffusion install directory must be added to the classpath for the VM:

- `../lib/diffusion.jar`
- `../etc`

Unless you set these properties programmatically when you instantiate your server, the `java` command that starts the server application must include the following arguments:

- `-Ddiffusion.license.file=`*`diffusion_installation`*`/etc/licence.lic`
- `-Ddiffusion.keystore.file=`*`diffusion_installation`*`/etc/publicKeys.store`
- `-Ddiffusion.home=`*`diffusion_installation`*`/lib`

Limitations

Currently only one Diffusion server can be instantiated in a Java VM and it can be started only once.

Concurrency

Diffusion is a multi-threaded server and utilizes concurrent processing to achieve maximum performance. Java NIO technology is utilized so that a separate thread is not required for each concurrent connection and very large numbers of concurrent connections can be handled.

Because Diffusion is a multi-threaded environment it is necessary to have an understanding of concurrency issues when writing publishers and when configuring Diffusion for best performance (see Tuning).

This section discusses issues of threading and concurrent processing.

Publisher threads

The processing that occurs within the user written code of a Publisher can be executed in different threads as discussed below. Any publisher method can be called at the same time as another. Because of this all publisher processing must be threadsafe and it is the user's responsibility to synchronize processing as required. It is recommended that synchronization is maintained at the smallest scope possible to avoid performance bottlenecks.

Inbound threads

Any input that is received on an NIO connection is processed by a thread from the inbound thread pool . This includes most publisher notifications from clients, event publishers or other publishers with the exception of control notifications (such as `initialLoad`, `publisherStarted`) which occurs in the controlling thread.

> **Note:** The act of publishing or sending messages to clients is asynchronous that is to say that the message is queued for the client or clients. Publisher processing is not blocked whilst messages are delivered to clients. For best performance it is recommended that any code executed in the inbound threads is non-blocking (for example, avoid database access, locking, and disk IO as much as possible).

Client notification threads

If a publisher uses Client notifications, the publisher has its own dedicated thread to process those notifications.

By default here is one notification thread per publisher, no matter how many listeners are defined. Each event is processed by the thread in the order in which they occur and two client notification event methods are not called concurrently. If the order of such events is not critical, you can specify that a user thread pool is used for client notifications this increasing throughput.

User threads

Publishers or other users of the Java API can make use of the Java threads API to schedule tasks for processing of their own in a separate thread of processing.

You can execute any object of a class that implements the `RunnableTask` interface using one of the `ThreadService.schedule` methods. You can to request a one-off execution of a task, periodic execution at a given interval or execution according to a schedule. Periodic processing can be important to publishers that pull data updates from elsewhere.

Such tasks issued using the thread service are executed using threads from the background thread pool.

Alternatively, users can define their own thread pools to use using the thread service and execute tasks using these thread pools.

NIO Threads

Each connector that is configured in `etc/Connectors.xml` comprises a connector thread and one or more acceptor threads. The connector thread listens for incoming socket connections, accepts them and registers them with an acceptor thread that handles any incoming data notifications. Message decoding, routing to publishers and appropriate publisher callbacks are all run in the inbound thread pool. Connector and acceptor threads are occupied for the minimum amount of time and are completely non-blocking.

Though performance can be improved in extreme case by adjusting the numbers of these NIO threads, no significant processing occurs within them.

Client multiplexers

A client multiplexer is a separate thread which is responsible for processing messages on the publisher event queue, queuing for clients (conflating if necessary), taking messages from client queues and sending them to the client or clients. A number of these multiplexers can be configured to improve concurrent processing when there are a large number of clients.

Multiplexers typically batch these output messages into output buffers according to the output buffer size configured for the client connectors.

Thread pools

Diffusion maintains a number of configurable thread pools which are used for a number of purposes

The detailed configuration and tuning of these pools is discussed in the tuning section. Thread pools can also be accessed programmatically using the `ThreadService` class of Diffusion server AP. Refer to the API documentation for more information about this.

The various types of thread pools are as follows:

Inbound thread pool

This is used to obtain a thread to process any inbound message received on an NIO connection. For example, clients and event publishers. The maximum number of threads configured for this pool must cater for the maximum required concurrency for incoming requests.

Diffusion does not maintain a separate thread for each client connection but rather passes each inbound request from a connection to the inbound thread pool for processing.

For example, when a client subscribes, the input processing happens on an inbound thread from the pool, the subscribe method and topic loader methods are run in one of these threads.

Connector inbound thread pools

Individual connectors can configure their own separate inbound thread pool to override the use of the default. This cannot be required if you want different behaviors for each connector or if there are a lot of connectors. Due to locking on the inbound thread pool, it is more performant for each connector to have its own inbound thread pool.

Background thread pool

The background thread pool is used for executing scheduled tasks. These tasks can be issued by Diffusion itself or using a publisher using the Java threads API.

Diffusion uses scheduled tasks for various reasons such as:

1. Timing out ACK messages. A scheduled task executes if a message sent to a client is not acknowledged within its required timeout period.
2. Retrying connections. If a Diffusion server cannot connect to another server and there is a retry policy, a scheduled task will be used to retry the connection.

If any publisher uses a lot of scheduled tasks, the number of threads in this pool might have to be increased waiting tasks might queue.

Unlike other types of pool when the specified number of threads are in use, tasks are queued in an unbounded queue.

User thread pools

Within the Java threads API user can define thread pools that can be used for multi-threaded processing.

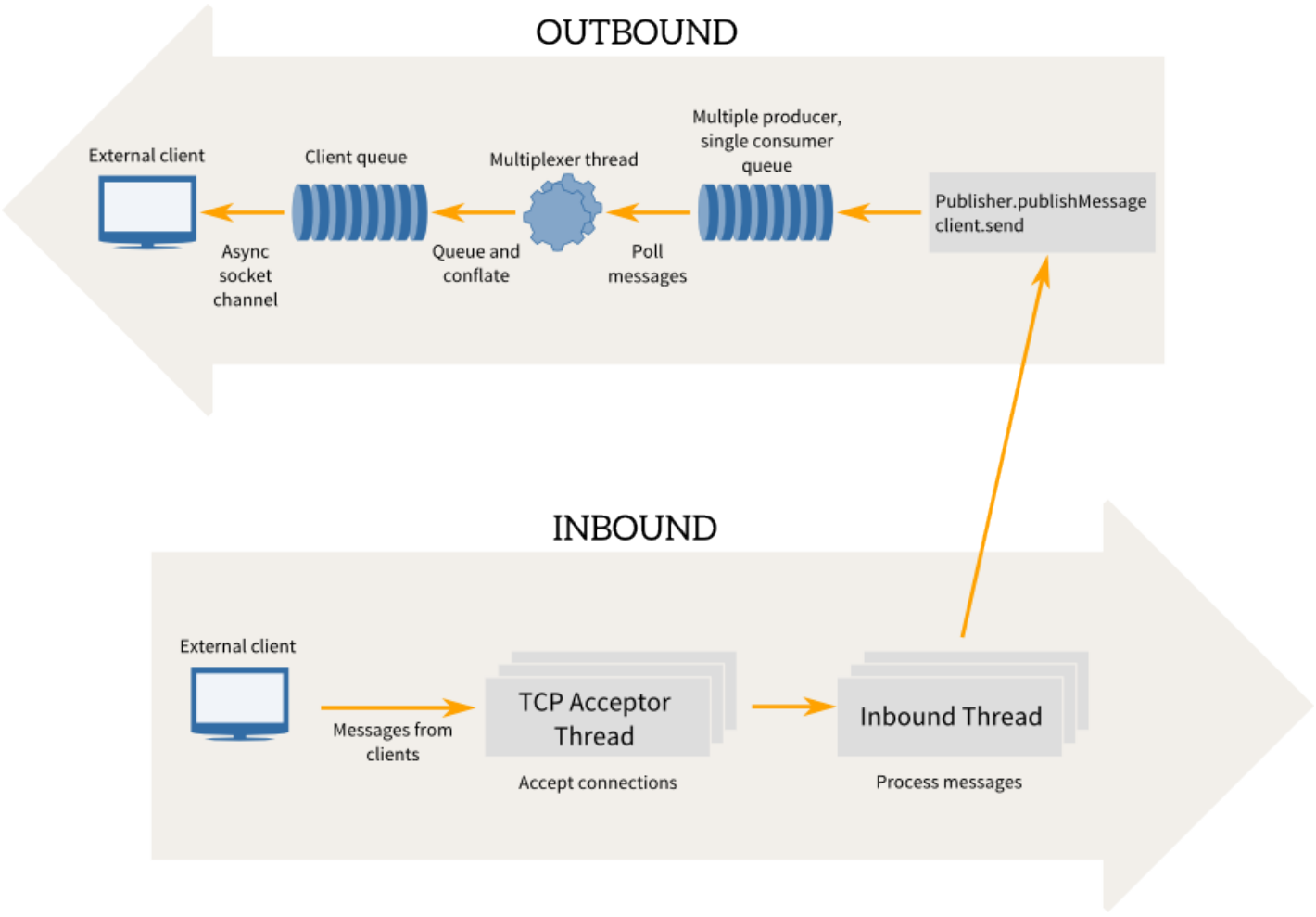


Figure 5: Thread diagram

Connectors

An introduction to the concept of connectors within a Diffusion server.

A connector provides a connection point for external applications to connect to a Diffusion server over a TCP connection.

Each connector has a 'Socket Server' thread which reacts to an incoming connection and passes it on to an acceptor to accept and process the connection. The socket information is defined by the connector.

Suitable connectors must be defined for inbound connections expected by a server. Connectors are defined in the `etc/Connectors.xml` configuration file.

The following properties are common to all connectors:

Table 7: Connectors properties

Name	A name by which the connector can be identified which will also be used for its acceptors.
Port	A port number on which to accept requests (or policy requests).
Host	The host to accept requests (only relevant on a multi-homed machine).
Number of acceptor threads	The number of acceptor threads. This can be tuned for performance reasons.
Input buffer size	The size of the socket input buffer to use for each connection.
Output buffer size	The size of the socket output buffer to use for each connection.

Socket buffer sizes are very important in achieving the best performance. See Tuning for more details.

Restricting connection types

By default a connector can accept any type of connection handled by Diffusion, but you can configure a connector so that it accepts only one type of connection as follows:

Table 8: Connection restrictions

client	Client connections.
policy	Policy file requests.
event	Event publisher connections.
all	Any type of connection.

Client connections

Connectors can accept connections from any type of client. Any number of connectors can be defined to provide different connection points with different properties.

Each client connection has an input buffer to receive messages from the client. The configured input buffer size must be large enough to accommodate the largest message expected from the client.

The output buffer size is used to assign an output buffer per client multiplexer into which messages are dequeued prior to transmission. The output buffer size must be at least as large as the largest message that is sent to the client (typically the largest initial topic load message). However, a much bigger size can be configured so that messages can be batched for sending to the client. Ideally the buffer size is a multiple of the average message size to allow for suitable batching. This can have an important effect on performance – see the Tuning section for more details.

Note: The buffer size must allow extra for header information when using HTTP.

Restricting client connections by API

Connectors can accept connections from clients that use either the Unified API or Classic API. Specify which API the connector accepts connection through by using the `<api-type>` element in the `etc/Connectors.xml` configuration file.

Policy connections

Connectors are used to serve policy file requests to plugin clients. Flash and Silverlight require policy files to be served from different ports (Flash on 843 and Silverlight on 943) so if plugin clients are in use it will be necessary to define a separate connector for each type of plugin client.

The connector configuration specifies the path of an XML policy file to be sent to the client.

Event publisher connections

The input buffer size specified for an event publisher connector is used to assign an input buffer in the same way as a client connector. It must be at least as large as the largest message expected.

The input buffer size specified can be much larger than the largest expected message size to allow for buffering and must match the size of the output buffer used by the event publisher itself.

The output buffer size specified must be able to accommodate the largest message that is sent to the event publisher (if any).

Related Links

Connectors on page 456
Connectors.xml - defines the connectors required.

Load balancers

Diffusion is commonly used with load balancers such as F5 Networks Inc.'s BIG-IP suite, to distribute incoming requests fairly over a number of Diffusion servers.

Each client is registered against a single Diffusion instance which contains client that is not replicated to other instances. For this reason, it is necessary for all traffic to and from that client is routed to the correct Diffusion server.

For HTTP-based protocols there might be many connections established, dropped and re-established during the lifetime of a single client session. Every one of these must be routed to the correct Diffusion server.

At first glance, streaming protocols such as DPT that open a single socket and remain connected until they are no longer required appear immune to requiring any special considerations. However, in the event that connection keep-alive is enabled to handle reconnections in case of temporary connection loss, it is important that the reconnection attempt is routed to the original server.

HTTP protocols

To route HTTP traffic, the load balancer must be able to inspect the HTTP headers and extract session information. Diffusion sets an HTTP cookie named `session` with a connection-specific ID specifically for this purpose. Load balancers typically have facilities for using this to maintain a table of client to server mappings.

Sample HTTP conversation, cookie highlighted:

```
POST /diffusion/ HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:22.0) Gecko/20100101
 Firefox/22.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-gb,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
m: 0
ty: B
t: null
tt: 90
v: 4
username: null
password: null
```

```
Referer: http://localhost:8080/tools/DhtmlClient.html
Content-Length: 0
Content-Type: text/plain; charset=UTF-8
Cookie: GUID=95UAaOVCK6FPFB2J9dBl
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache

HTTP/1.1 200 OK
Set-Cookie: session=clyde-r6t80mwyjav6
Cache-Control:no-store, no-cache
Content-Type:text/plain; charset=UTF-8
Content-Length:26

4.100.4.clyde-r6t80mwyjav6

POST /diffusion/ HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:22.0) Gecko/20100101
 Firefox/22.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-gb,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
m: 1
c: clyde-r6t80mwyjav6
Referer: http://localhost:8080/tools/DhtmlClient.html
Content-Length: 0
Content-Type: text/plain; charset=UTF-8
Cookie: session=clyde-r6t80mwyjav6; GUID=95UAaOVCK6FPFB2J9dBl
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
```

Streaming protocols

Without the ability to parse headers (and indeed, the absence of a session ID at all), the most common method for routing a streaming protocol such as DPT or websockets is to create a client/server mapping based on the IP addresses of the endpoints. This technique is generally referred to as Sticky-IP, and has the advantage of also working with HTTP transports, if required.

For F5's Sticky IP, ensure that the Source Address Translation option is set to Auto Map.

Figure 6: Sticky-IP in F5 BIG-IP

The drawback of this approach is that multiple users masquerading behind a proxy or access point can have the same IP address, and all requests from clients with that IP address are routed to the same Diffusion instance. Load balancing still occurs, but some hosts might be unfairly loaded.

Routing strategies

Load balancers often present several different strategies for choosing which server is associated with a new request. Common examples are:

Table 9: Routing strategies

Name	Description
Round-robin	Each available Diffusion instance is chosen in turn, with none favored.
Fewest clients	The server with the fewest number of client connections in progress is chosen.
Least loaded	The server with the lowest CPU load is chosen.

Monitors

Determining the availability of a Diffusion node can be achieved in more than one way. Commonly, an HTTP probe is used against the built-in web server. This has the advantage of being simple; most system administrators are familiar with HTTP requests. In the simplest case, a `GET` request can be made against the root context of the web server, for example:

```
GET / HTTP/1.0\r\n
```

However, this only tests the availability of the Diffusion server as a whole, and not the applications within it. A more advanced approach is to implement an `http-service`, as documented in the web-servers section. This queries the state of the publishers or the topic tree, and return availability on receipt of a `GET` request.

A better, but more complex approach is to implement a custom monitor using a scripting language that is supported by the load balancer. In the case of BIG-IP, this is a custom Diffusion client written in Perl that connects and subscribes to a status topic. Details on the Diffusion protocol can be found in the Protocol chapter.

> **Note:** Push Technology does not provide or support custom monitors written by third parties.

Connection pooling

Many load balancers include a connection pooling feature, such as the BIG-IP OneConnect profile, where connections between the load balancer and the Diffusion server are kept alive and reused by other clients. In fact, multiple clients can be multiplexed through a single server side connection.

In Diffusion, a client is associated with a single TCP/HTTP connection for the lifetime of that connection. If a Diffusion server closes a client, the connection is also closed. Diffusion makes no distinction between a single client connection and a multiplexed connection, so when a client sharing a multiplexed connection closes, the connection between the load balancer and Diffusion is closed, and subsequently all of the client-side connections multiplexed through that server-side connection are closed.

For this reason, it is required that load balancers are not configured to pool connections when working with Diffusion.

Further reading

F5/BIG-IP:F5: Enabling Session Persistence

Replication

Diffusion uses a datagrid to share session and topic information between Diffusion servers and provide high availability for clients connecting to load-balanced servers.

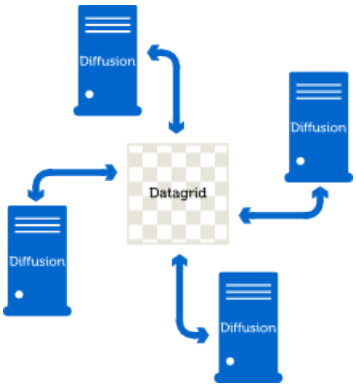


Figure 7: Information sharing using a datagrid

Diffusion uses Hazelcast™ as its datagrid. Servers reflect session and topic information into the datagrid.

If a server becomes unavailable, another server can access the session and topic information that is stored in the datagrid and take over the responsibilities of the first server.

Session replication

You can use session replication to ensure that if a client connection fails over from one server to another the state of the client session is maintained.

When a connection from a client through the load balancer to a Diffusion server fails, the load balancer routes the client connection to another Diffusion server. This server has access to the session and client information that is replicated in the datagrid.

Control clients connect to a specific Diffusion server and not through a load balancer. Session replication cannot be used for control clients.

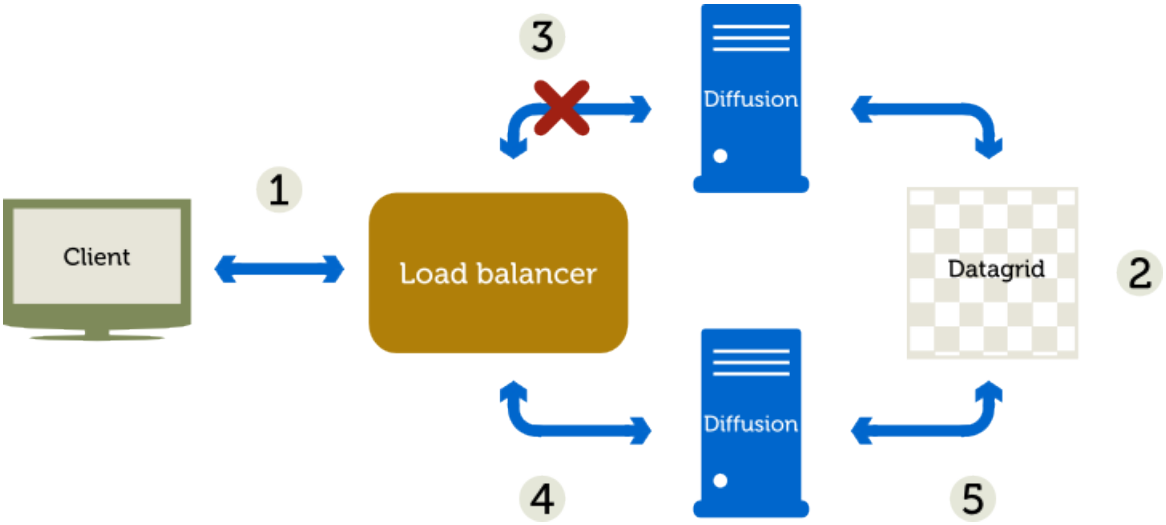


Figure 8: Session replication

1. A client connects to a Diffusion server through a load balancer. The load balancer is configured to route based on the client's session ID and requests from the client go to the same server until that server becomes unavailable.
2. Information about the client, such as its credentials, session ID, and subscription information, is stored in the datagrid.

3. A client loses connection to the Diffusion server if the server becomes unavailable.
4. The client can reconnect and the load balancer routes the connection to another Diffusion server.
5. This Diffusion server has access to all of the client information shared into the datagrid by the first Diffusion server. It provides initial topic load messages for all topics the client subscribes to.

The client can reconnect to its session only if it reconnects within the reconnect time specified in the `Connectors.xml` configuration file. If the client does not reconnect within that time, the client session information is removed from the datagrid.

Considerations

Consider the following factors when using session replication:

- Messages in transit are not preserved. Use acks to ascertain whether or not messages have been received.
- The failover appears to the client as a disconnection and subsequent reconnection. To take advantage of high server availability, clients must implement a reconnect process.
- Replication of session information into the datagrid is not automatic. It must be configured at the server.

Related Links

replication on page 473
Replication.xml - defines the replication performed by the server.

Client reconnection on page 504

You can configure the client reconnection feature by configuring the connectors at the Diffusion server to keep alive the client session.

Topic replication

You can use topic replication to ensure that the structure of the topic tree, topic definitions, and topic data are synchronized between servers.

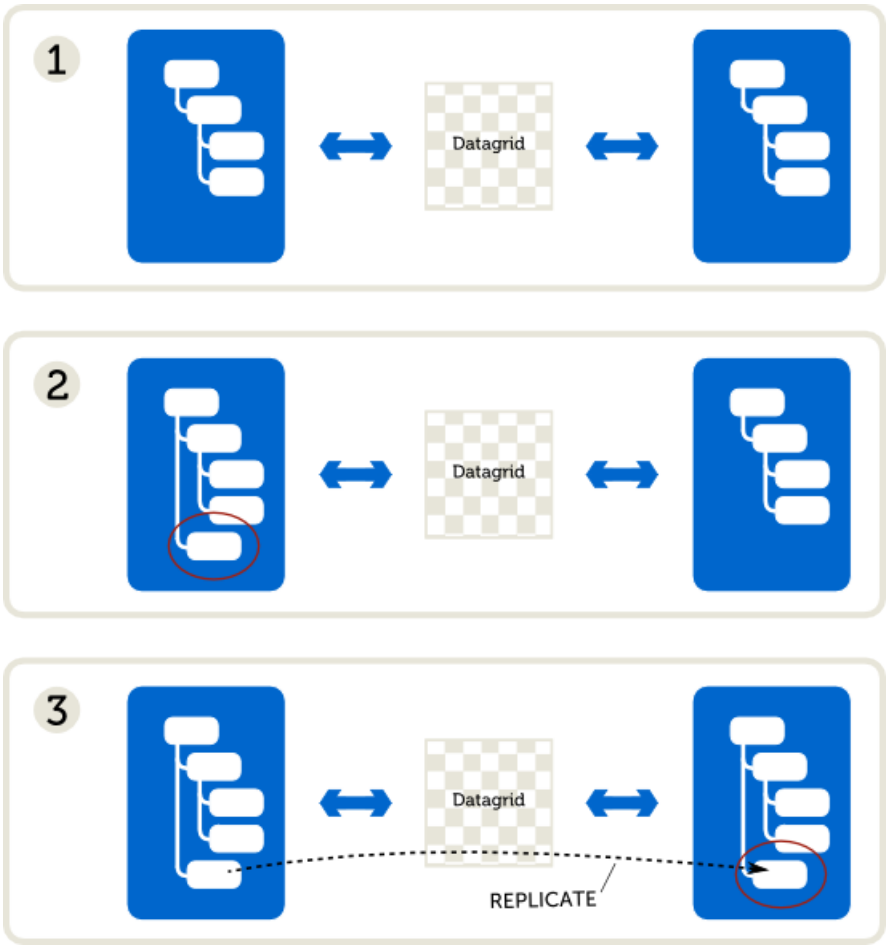


Figure 9: Topic replication

1. Servers with topic replication enabled for a section of the topic tree share information about that section of the topic tree through the datagrid. The topic information and topic data are synchronized on all the servers.
2. A new topic is created on one server in the replicated section of the topic tree.
3. The new topic is replicated on the other servers with identical topic information. When its topic data is updated on the first server, that data is replicated on the other servers.

Considerations

Consider the following factors when using topic replication:

- Only publishing topics can be replicated.
- Replication is supported only for the following types of topic data:
  - No topic data
  - Custom topic data
  - Protocol buffer topic data
  - Record topic data
  - Single value topic data
- Replication is not supported for paged topics.

- Only topic-wide messages are replicated. Messages sent to a single client or to all clients except one are not replicated.
- Replication of topic information into the datagrid is not automatic. It must be configured at the server. This gives a performance advantage, as you can choose which parts of your topic tree to replicate.
- Replication of topic data can impact performance.
- Do not use topic replication on sections of the topic tree that are owned and updated by publishers. Topic updates sent by publishers are not replicated.
- Avoid including replicated topics in REMOVE_TOPICS session wills. When a replicated topic is removed from a server as a result of a session will, it is removed from all other servers that replicate that topic. If all sessions on a Diffusion server that have a remove topics removal request for a branch of the topic tree close, the topics are removed even if that topic is replicated and sessions on other Diffusion servers have removal requests registered against that part of the tree. When the topics are removed on the server, that change is replicated to all other servers that participate in replication for these topics.

Related Links

replication on page 473
Replication.xml - defines the replication performed by the server.

Failover of active update sources

You can use failover of active update sources to ensure that when a server that is the active update source for a section of the topic tree becomes unavailable, another server is assigned to be the

active update source for that section of the topic tree. Failover of active update sources is enabled for any sections of the topic tree that have topic replication enabled.

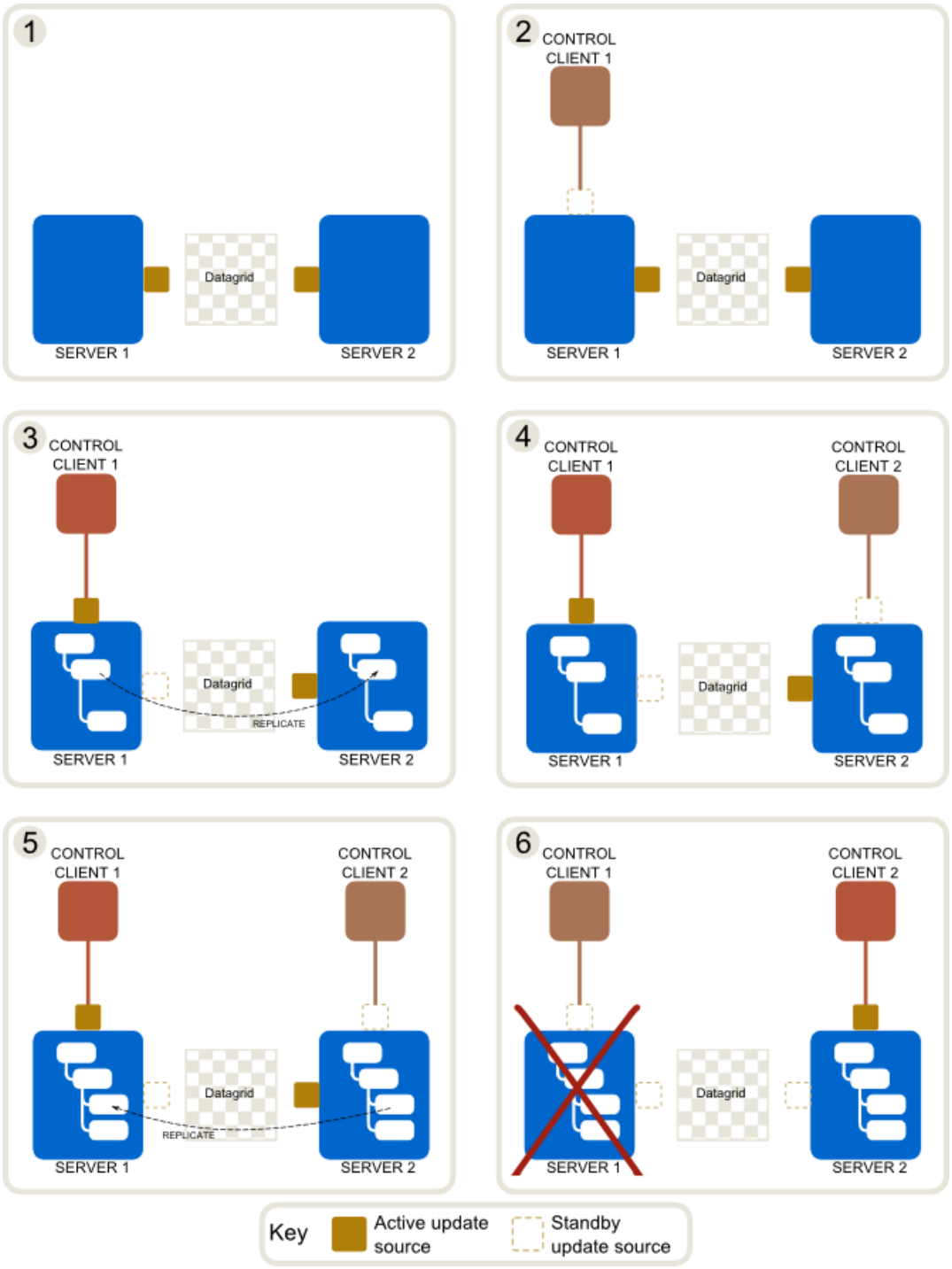


Figure 10: Failover of an active update source

1. Servers that have topic replication enabled for a section of the topic tree create update sources for that section of the topic tree when the server is started.

   These active update sources point into the datagrid and are ready to receive any updates for topics in that section of the topic tree that are reflected into the datagrid.
2. CONTROL CLIENT 1 starts and registers an update source for that section of the topic tree with SERVER 1. This update source is a standby update source. The update source that points at the datagrid remains the active update source.
3. CONTROL CLIENT 1 creates a topic within that section of the topic tree. Through topic replication, the new topic is also created on SERVER 2.

The update source that SERVER 1 directs at the datagrid becomes a standby update source and the update source that CONTROL CLIENT 1 registered with SERVER 1 becomes active. SERVER 1 sends CONTROL CLIENT 1 a callback to notify it that it is the active update source.

On SERVER 1, the topics in that section of the topic tree receive their updates from CONTROL CLIENT 1. SERVER 1 reflects this topic data into the datagrid. On SERVER 2, the topics in that section of the topic tree receive their updates from the datagrid.

4. CONTROL CLIENT 2 starts and registers an update source for that section of the topic tree with SERVER 2. The update source for CONTROL CLIENT 2 remains a standby update source and the active update source for SERVER 2 remains pointing at the datagrid.

   The topics on SERVER 2 continue to receive their updates from the datagrid.
5. On SERVER 2, CONTROL CLIENT 2 (or another publisher or control client) can create topics in that section of the topic tree. This does not affect the active update source for that section of the tree.

   The new topic is reflected into the datagrid. Through topic replication, SERVER 1 replicates the new topic in its topic tree. CONTROL CLIENT 1 is the active update source for the new topic.
6. If SERVER 1 becomes unavailable, SERVER 2 becomes the owner of that section of the topic tree and the update source registered by CONTROL CLIENT 2 becomes active. SERVER 2 sends CONTROL CLIENT 2 a callback to notify it that it is the active update source.

   On SERVER 2, the topics in that section of the topic tree receive their updates from CONTROL CLIENT 2. SERVER 2 reflects this topic data into the datagrid.

Considerations

Consider the following factors when using failover of active update sources:

- If the topic paths that the control client uses to register an update source do not match the topic paths configured in the `Replication.xml` configuration file of the server, unexpected behavior can occur.
- Failover of active update sources is not automatic. It must be configured at the server.
- The mechanism that provides failover of active update sources assumes that all servers have the same configuration and that all control clients implement the same behavior as part of a scalable and highly available deployment. If this is not the case, unexpected behavior can occur.
- Do not use topic replication and failover of active update sources on sections of the topic tree that are owned and updated by publishers. Topic updates sent by publishers are not replicated.

Related Links

replication on page 473
Replication.xml - defines the replication performed by the server.

Updating topics from a control client on page 234
A control client can use the TopicUpdateControl feature of the Unified API to update topics.

Configuring replication

You can configure replication by editing the `etc/Replication.xml` files of your Diffusion servers.

Procedure

1. Edit the `Replication.xml` file to configure replication.

```
<replication enabled="true">
    <provider>HAZELCAST</provider>
    <sessionReplication enabled="true" />
    <topicReplication enabled="true">
        <topics>
            <topicPath>foo/bar</topicPath>
```

```
        </topics>
    </topicReplication>
</replication>
```

- In the `replication` element, set `enabled` to `true` to enable replication.
- In the `sessionReplication` element, set `enabled` to `true` to configure the server to reflect client session information into the datagrid.
- In the `topicReplication` element, set `enabled` to `true` to configure the server to reflect topic information and topic data into the datagrid.
- Inside the `topics` element, use one or more `topicPath` elements to define the topics to which to apply topic replication and failover of the active update source.

  The content of the `topicPath` element is a path to a single topic. Topic replication and failover of the active update source are applied to the topic defined by the path and all topics below it in the topic tree.

  Unlike a topic selector, the topic path does not contain any leading or trailing characters. For example, use `<topicPath>foo/bar</topicPath>` to select the topic `foo/bar`.

  The `topicPath` elements also define which sections of the topic tree are have update sources created for them by the server.

2. Restart the Diffusion server to load the configuration.
3. Ensure that your clients are configured to reconnect if they lose their connection to the server.

Related Links

replication on page 473
Replication.xml - defines the replication performed by the server.

Configuring your datagrid provider

You can configure your datagrid to work correctly with your solution architecture.

Configuring Hazelcast

By default, the Hazelcast node in your Diffusion server multicasts to all other Hazelcast nodes in your network.

To define which Hazelcast nodes can communicate with each other, use the `hazelcast.xml` configuration file. For example, you can configure the Hazelcast nodes in the development environment to communicate only with other nodes in the development environment and not with Hazelcast nodes in the production environment.

The following example shows the structure of the `hazelcast.xml` file:

```
<hazelcast xsi:schemaLocation="http://www.hazelcast.com/schema/config
    http://www.hazelcast.com/schema/config/hazelcast-config-3.0.xsd"
    xmlns="http://www.hazelcast.com/schema/config" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance">

    <properties>
        <property name="hazelcast.logging.type">slf4j</property>
    </properties>

    <network>

        <join>
            <!-- <multicast enabled="true" /> -->
            <tcp-ip enabled="true">
             <member>node1.example.com</member>
                <member>203.0.113.1</member>
                <member>203.0.113.2:5757</member>
                <member>203.0.113.3-7</member>
            </tcp-ip>
        </join>
```

```
    </network>
</hazelcast>
```

This example configuration disables the multicast capability and defines the Hazelcast nodes that can be connected to.

The Hazelcast nodes can be defined by hostname, by IP address, or by IP range. The default port used by Hazelcast is 5701. If you want to connect on a different port, you can specify this when you define the node, using the format *host*:*port*.

Ensure that the `hazelcast.xml` file is on the Diffusion server classpath. For example, by putting the file in the *`diffusion_installation`*`/etc` directory. Restart the Diffusion server to load the configuration.

For more information about using the `hazelcast.xml` file to configure Hazelcast, see the Hazelcast™ Reference Manual.

Diagnosing problems with Hazelcast

If you enable logging for Hazelcast, you can use the log files to diagnose problems with Hazelcast.

To enable logging, include the following line in your `hazelcast.xml` file:

```
<property name="hazelcast.logging.type">slf4j</property>
```

Ensure that the `hazelcast.xml` file is on the Diffusion server classpath. For example, by putting the file in the *`diffusion_installation`*`/etc` directory. Restart the Diffusion server to load the configuration.

You can also enable logging by starting the Diffusion server that contains the node with the following parameter `-Dhazelcast.logging.type=slf4j`

You can enable JMX for your Hazelcast nodes and use a JMX tool to examine the MBeans.

To enable JMX for a Hazelcast node, include the following line in your `hazelcast.xml` file:

```
<property name="hazelcast.jmx">true</property>
```

Ensure that the `hazelcast.xml` file is on the Diffusion server classpath. For example, by putting the file in the *`diffusion_installation`*`/etc` directory. Restart the Diffusion server to load the configuration.

You can also enable JMX by starting the Diffusion server that contains the node with the following parameter `-Dhazelcast.jmx=true`

For more information about using Hazelcast, see the Hazelcast™ Reference Manual.

Chapter

3

Web server

In this section:

- Interaction with publishers
- Security

Diffusion incorporates its own web server.

Any Diffusion connector can be configured to act as a web server and service Diffusion clients, file requests, or any request for which a user-defined HTTP service has been configured.

Interaction with publishers

Describes how publishers can interact with the web server

Server-side processing

Server-side processing can be utilized with any file that has a text mime type and JavaScript. Currently there are three server-side tags, Include, Publisher and Topic Data. These tags are stored in HTML comments so as to not interfere with normal HTML

Include Tag

Include stubs load the file specified in the file attribute and are loaded as is into the parent HTML document. They do not necessarily have to be valid HTML. They can be positioned anywhere within the HTML file.

These includes are synonymous with `#Include` statements of ANSI C.

Below is an example of the syntax:

```
<!--@DiffusionTag type="Include" file="stub.html" -->
```

Include files can be nested so an include file can contain an include tag

Publisher tag

Publisher tags enable a publisher to interact with the web page during the serving process. Again these tags can appear anywhere within the HTML document. In the case below the publisher method `processHTMLTag` of the `Trade` publisher is called with the tag argument of `table` The publisher can return a String of HTML that is inserted into the document at the position of the tag and the tag is removed. The `processHTMLTag` method is also called with the HTTP Request, although the request cannot be written to. Below is an example of the syntax

```
<!--@DiffusionTag type="publisher" publisher="Trade" tagid="table" -->
```

TopicData

Topic data tags allow for `SingleValueTopicData` items to be rendered in the HTML page. Again these tags can appear anywhere within the HTML document. The following example shows the syntax:

```
<!--@DiffusionTag type="TopicData" name="Assets/FX/EURUSD/O" -->
```

HTTP listener

Publishers can listen to all file HTTP requests by registering as a `HTTPRequestListener`. This exposes the interface

```
void handleHTTPRequest(HTTPVirtualHost virtualHost,HTTPRequest request)
```

This enables for more detailed statistics to be captured from the HTTP request

Security

Aspects of web server security

Digest authentication

Digest authentication can be utilized to negotiate credentials with a user's web browser. It is applied to specific directories on your web site. The protection of one directory automatically applies protection to all lower directories as well. The configuration file `etc/WebServer.xml` is used to add new realms and to store the user's name and the passwords.

HTTP deployment

You can deploy DAR files to a Diffusion server through a web service. This web service does not run by default, but can be enabled for your test environment by editing the `WebServer.xml` configuration file.

> **Warning:** Access to the deploy web service is not restricted. Do not enable this web service in your production environment unless you restrict access to the *`diffusion-url`*`/deploy` URL by other means, for example through your firewall setup.

Chapter 4

Publishers

In this section:

- Publisher basics
- Writing a publisher
- Testing a publisher
- Client queues
- Client validation
- Client Geo and WhoIs information
- Client groups
- Client notifications
- Design patterns

A publisher lies at the core of the Diffusion infrastructure. A publisher publishes messages to clients that are subscribed to the topics provided by the publisher.

There can be one or more publishers implemented within a Diffusion server. A publisher can provide the behavior of one or more topics but a topic can belong to only one publisher. The publisher infrastructure is provided by Diffusion and the behavior is provided by the user by writing a publisher. A publisher is written in Java and deployed within the Diffusion server itself. The following sections describe in more detail how to write a publisher and describe some of the facilities available to a publisher.

Publisher basics

A publisher is a user-defined object deployed within a Diffusion server which provides one or more topics on which it publishes messages to clients.

There can be one or more publishers deployed with a Diffusion server.

Clients connect to the server and subscribe to topics. Messages relate to topics and when a publisher publishes a message it is broadcast to all clients that are currently subscribed to the message topic. A publisher can also send messages to individual clients and receive messages sent from clients. Clients can request (fetch) topic state, even when not subscribed.

A publisher must be written by the user in Java (utilizing the publisher API) and deployed within the server. This is done by extending a supplied publisher class and implementing methods as required. Implement the methods relating to the functionality that you require. For more information, see Writing a publisher on page 87.

Defining publishers

How to define publishers that start with Diffusion.

The Diffusion server is able to start the publishers defined in the `etc/Publishers.xml` file when the server starts. The XML file can contain any number of publishers. Each publisher must have at least a name and a class. The class must implement the publisher by extending the `Publisher` class For more information, see 'Creating a Publisher class on page 88.

```
<publishers>
 <publisher name="Publisher">
  <class>com.example.Publisher</class>
 </publisher>
 ...
</publishers>
```

The name must be unique on the server, and the class must exist on the classpath of the Diffusion server (See Classic deployment). This is sufficient for the publisher to start when Diffusion does. There are other options, including those that can prevent the publisher from starting.

When the `enabled` element is false, the publisher class is not loaded. If the `start` element is false, the publisher is not started when the server starts. If the `topic-aliasing` element is false, topic aliases is not used by topic messages. The following example shows the default values for these optional settings.

```
<publishers>
 <publisher name="Publisher">
  <class>com.example.Publisher</class>
  <enabled>true</enabled>
  <start>true</start>
  <topic-aliasing>true</topic-aliasing>
 </publisher>
</publishers>
```

You can define properties in the `etc/Publishers.xml` that can be accessed from the publisher, see Publisher Properties for details.

See the Publisher client connection section to see how server connections can be configured in the `etc/Publishers.xml` for connecting to other publishers.

The full configuration file options can be found in the XSD document for the `etc/Publishers.xml` or XML Configuration – publishers.

Loading publisher code

This describes how to load publisher classes or code it is dependent upon.

When you write a publisher class (or any other classes it uses), you can deploy them in any folder as long as it is specified in the configuration (usr-lib in `etc/Server.xml`). JAR files can also be deployed in user libraries and any other software libraries that the publisher requires can be specified in this way.

Also, when Diffusion starts, the `data` and `etc` folders are on the class path. The `ext` folder, and its sub-directories are scanned for jar files and class loaded. This means that you can easily add new jars to the Diffusion runtime, without having to edit the startup scripts.

Take care when creating backup jars in the `ext` folder as anything that ends in `.jar` is class loaded.

Programmatically loading publishers

You can configure and load custom publishers using the Diffusion API at any point in the server's lifecycle.

Similarly to loading publishers using configuration files, each publisher must have at least a name and a class. The class must implement the publisher by extending the `Publisher` class (See Creating a Publisher class)

```
PublisherConfig config =
 ConfigManager.getServerConfig().addPublisher("MyPublisher",
 "com.acme.foo.MyPublisher");
Publisher publisher = Publishers.loadPublisher(config);
```

The name must be unique on the server, and the class must exist on the classpath of the Diffusion server. For more information, see Classic deployment on page 382. By default the `autostart` property is enabled on the PublisherConfig, so the publisher starts once it is loaded. If this option is disabled, you can load a publisher and retain a reference to it, to start at a later point in time.

If the default configuration options are suitable for your requirements (as detailed within the API docs for `com.pushtechnology.diffusion.api.config.PublisherConfig`) there are several convenience methods that can be used to load a given publisher and get a reference to it without the need for construction a specific `PublisherConfig` instance.

```
// Create Publisher with classname
Publisher publisher = Publishers.createPublisher("MyPublisher",
 "com.acme.foo.MyPublisher");

// Create Publisher with Class
Publisher publisher = Publishers.createPublisher("MyPublisher",
 MyPublisher.class);
```

You can load a default publisher instance. This facilitates programmatic access any features exposed through the publisher abstract class that do not require method overriding.

```
Publisher publisher = Publishers.createPublisher("MyDefaultPublisher");
```

Starting and stopping publishers

Typically publishers are started when the server starts but you can prevent such automatic start up and allow publishers to be started using System Management.

Publishers can also be stopped and restarted using System Management functions and are automatically stopped and removed when the server closes.

In order for a publisher to function properly on being stopped and restarted from System Management it must be able to cater for the integrity of its data and client connections. For this reason a publisher cannot be stopped by default and must override the `isStoppable` method to enable this functionality.

Publisher startup steps

When a publisher is started it goes through its initial processing in the order shown below:

Table 10: Start publisher

Add initial topics	Initial topics configured for the publisher are added.
Load server connections	Server connections configured for the publisher are loaded and validated.
`initialLoad`	The `intialLoad` notification method is called. This can be used to perform any initial processing required for the publisher. Topics can be added here. Other aspects of the publisher, such as topic loaders and client listeners can also be set up here. If an exception is thrown by this method, the publisher fails to start.
Connect to servers	A connection is made to each server connection .
STARTED	At this point the publisher is considered to have started.
`publisherStarted`	The `publisherStarted` notification method is called.

Publisher closedown steps

When a publisher is stopped, either during server closedown or by System Management it goes through the following steps:

Table 11: Stop publisher

`publisherStopping`	The `publisherStopping` notification method is called to allow the publisher to perform any preliminary close processing.
Remove topics	All topics owned by the publisher are removed.
Close server connections	Any server connections made by the publisher are closed.
STOPPED	At this point the publisher is considered to be stopped.
`0publisherStopped`	The `publisherStopped` notification method is called.
Client events Stopped	Client event notifications are stopped.

Publisher removal

A publisher is removed after it is stopped during server closedown but you can also remove a stopped publisher at any time using System Management. Once removed a publisher cannot be restarted again until the server is restarted.

In either case, after removal the `publisherRemoved` notification method is called.

Publisher topics

Topics are the mechanism by which publishers provide data to clients.

Each publisher can provide one or more topics but each topic must be unique by name within the server. Topics are hierarchical in nature and so topics can be parents of topics and a tree of topics can be set up. Using hierarchies allows clients to subscribe to branches of the hierarchy rather than having to subscribe to individual topics. Only the owner of a topic can create new topics below it in the hierarchy.

Adding topics

In the simplest case a publisher can name the topics it provides within its configuration. In this case such topics are automatically added as the publisher is started. These topics can be obtained from within the publisher using the `getInitialTopicSet` method.

More typically a publisher adds the topics it requires itself as it starts up. A Publisher can choose to add some topics at start up and others later. Topics can be added at any time using the publisher's `addTopic` or `addTopics` method. They can be added only if they are added by the owner of the parent topic.

A topic can be a simple topic where all of the handling of the topic state is provided by the publisher. Alternatively a topic can be created with topic data which handles the state of the topic automatically.

As soon as a topic has been added clients can subscribe to it.

Loading topics

Simple topic processing involves sending all of the data that defines a topic (the topic load) to a client when they first subscribe and then subsequently sending deltas (or changes to the data). There are two mechanisms for performing the topic load:

Send on subscribe
: When the publisher is notified of subscription it creates, populates and sends a topic load message to the client.

Topic loaders
: Define a topic loader for the topic which is automatically called to perform the topic loading when a client subscribes.

If a topic has topic data, the current state is automatically provided to a client when they subscribe.

Subscribing clients to topics

Clients normally request subscription to a topic and if the topic exists the clients become subscribed to it at that point.

A client can subscribe to a topic that does not exist at that time – this is called pre-emptive subscription. In this case, when a publisher creates a new topic it can subscribe any clients that have pre-emptively subscribed to it by using the subscribeClients method specifying `force=false` or this happens automatically if the parent topic level has automatic topic subscription specified

A publisher can also force all currently connected clients to become subscribed to a topic by calling `subscribeClients` with `force=true`.

Subscribing clients to topics that they were already subscribed to causes the topic load to be performed again.

A publisher can also cause individual clients to be subscribed to a topic using the client's subscribe method or unsubscribed using the unsubscribe method.

Providing topic state

The publisher method `fetchForClient` must be implemented if clients are to obtain state using the topic fetch feature.

Handling topics that do not exist

A topic is an entity that notionally has state but in some circumstances a client might request access to a topic that does not exist. Client notifications provide a mechanism whereby this situation can be handled.

Where a client attempts to subscribe to a topic that does not exist, a `clientSubscriptionInvalid` notification occurs which gives the publisher the opportunity to dynamically create the topic (and subscribe the client to it) if that is what is required.

Where a client attempts to fetch the state of a topic that does not exist, a `clientFetchInvalid` notification occurs which gives the publisher the opportunity to return a response to the fetch request (using sendFetchReply) even if the topic does not exist. This can provide an efficient request/response mechanism without the overhead of actually creating topics.

If a client attempts to send a message to a topic to which it is not subscribed, including the case where that topic does not exist, a `clientSendInvalid` notification occurs, allowing the publisher to either create, subscribe the client, process the message as stand-alone data or discard the message as appropriate.

Removing topics

A publisher can also remove topics at any time using its `removeTopic` or `removeTopics` methods. Removing a topic also removes any topics beneath it in the topic hierarchy.

Removing a topic causes all clients that are subscribed to it to be unsubscribed.

Receiving and maintaining data

A publisher can obtain the data it is to publish and transform that data in any way that is appropriate.

The publisher maintains the state of its own data by updating it whenever any changes are received so that as a new client subscribes it can be sent the latest state of the data as a whole. As such changes are received they are also published as deltas to all currently subscribed clients.

Receiving messages from an event publisher

A typical way of receiving data is from an event publisher where messages for publisher topics are routed to the publisher through its `messageFromEventPublisher` method.

If a publisher uses event publishers, it must add a listener for event publisher connection events using `Publishers.addEventListener`. The publisher can implement EventConnectionListener and add itself or it can use some other object. This interface allows the publisher to be notified whenever event publishers connect or disconnect.

Receiving messages from a remote service

Remote service can also provided a data feed into a publisher. The remote service can publish to topics and the updates are applied to the topics and passed onto subscribed clients.

Publishing and sending messages

Publishing messages to clients and sending messages to clients

Creating messages

Messages can be created using the factory methods on the publisher or on a topic for creating messages (called `createLoadMessage` and `createDeltaMessage`).

If within a class that does not have a direct reference to the publisher or topic objects, the equivalent static methods in the `Publishers` class can be used. Messages can be populated with data using the many variants of the `put` method.

Publishing messages

Messages (whether load or delta) can be sent to all clients that are subscribed to the message topic using the publisher's `publishMessage` method.

Messages can be sent to an identified group of clients using client groups.

Exclusive publishing

You might want to publish a message to all but a particular client. For example, a message can be sent to the publisher from a client and the publisher can, publish the message to all of the other subscribed clients.

This is done using the publisher's `publishExclusiveMessage` method. This facility also exists on client groups.

Sending messages to individual clients

To send a message to an individual client the `Client.send` method can be used.

To send a message to a group of clients the `ClientGroup.send` method can be used.

Publisher notifications

A publisher is notified of certain events by certain methods on it being called. These methods can be overridden by the user to perform processing at these points as required.

By default these methods (other than those indicated) perform no processing. You do not have to override any of these methods unless you choose to. The notification methods are:

Table 12: Notification methods

`initialLoad`	Called when the publisher is first loaded. Is typically overridden to perform any initial processing required to prepare the publisher.
`publisherStarted`	Called after initialLoad (see startup steps).
`subscription`	Called when a client subscribes to a topic that the publisher owns. References to the topic and the client are passed and also a flag to indicate if the topic has already been loaded by a TopicLoader. If the topic has not been loaded already, typically a publisher sends an initial load message to the client at this point. It might not be necessary to override this method if topic loaders are in use.
`unsubscription`	Called when a client unsubscribes from a topic that the publisher owns.
`messageFromClient`	Called when a message is received from a client. References to the message and the client are passed.
`messageFromEventPublisher`	Called when a message is received from an event publisher. References to the message and the event publisher are passed.

`messageFromServer`	Called when a message is received from a server connection. References to the message and the server connection are passed.
`fetchForClient`	Called when a client requests a fetch of the topic state for a topic that does not have a `TopicFetchHandler`declared and does not use TopicData.
`messageNotAcknowledged`	Called when a message which required acknowledgment was sent by the publisher and was not acknowledged by one or more clients within the given timeout period.
`serverConnected`	This is called when a server connection is made. A reference to the server connection is passed.
`serverTopicStatusChanged`	This is called when a topic subscribed to at a `ServerConnection` has its status changed (for example, is removed). The topic name and reference to the server connection is passed.
`serverDisconnected`	This is called when a ServerConnection is lost. A reference to the server connection is passed.
`publisherStopping`	This is called when the publisher has been requested to stop. It gives the publisher the opportunity to tidily perform any close processing.
`publisherStopped`	This is called after a publisher has stopped. The publisher can still be restarted (but only if `isStoppable` is true).
`publisherRemoved`	This is called when a publisher is removed and provides the opportunity for final tidy up. The publisher cannot be restarted after this is called.
`systemStarted`	This is called when the Diffusion system has completed loading and is ready to accept connections. Publishers are started before connectors, so this notification is used all Diffusion sub systems are loaded.

Publisher notification threads

To understand issues of concurrency when writing a publisher it is necessary to understand in which threads the various publisher notifications occur.

When a message or request is received from a client (of any sort), event publisher or server connection, the inbound thread pool is used to process it. Depending upon the number of threads in the pool this can mean that the publisher can receive such notifications concurrently.

Message acknowledgment notifications use the background thread pool.

Other notifications come from various control threads.

All of the above considerations mean that concurrency must always be taken into account in publisher code and it must be made threadsafe as appropriate.

Client handling

A publisher can receive notifications about and perform actions on individual clients.

Closing/Aborting clients

A publisher can close a client at any time using the close method. This disconnects the client which might choose to subsequently reconnect.

Alternatively a publisher can use the abort method, which sends an abort notification to the client before disconnecting it. A client receiving an abort notification must not attempt to reconnect.

Client notifications

A publisher can choose to receive additional client notifications so that it can be informed when clients connect, disconnect etc.

Client pings

A client ping message is one that can be sent to a client which reflects it back to the server to measure latency. A publisher can send a ping message to a client using the `Client.ping` method and receives a response on the `ClientListener.clientPingResponse` method within which the message passed can be queried to establish the round trip time.

Client message filtering

You can filter the messages that get queued for any particular client. For more information, see Message filters on page 214.

Publisher properties

Properties for a publisher are defined in the `etc/Publishers.xml` configuration file.

As well as the standard properties a publisher can have user-defined properties. These properties can be read using convenience methods available on the publisher (for example, `getProperty`, `getIntegerProperty` etc).

Using concurrent threads

Often within a publisher you might have to initiate some processing in a separate thread so that the publisher itself is not blocked.

For example, a thread can be used to poll data from some data source.

Diffusion provides a mechanism for easily managing concurrent processing using the threads API.

Publisher logging

Every publisher is assigned its own Logger which can be used within the publisher itself for logging diagnostic messages.

This Logger is obtained using the getLogger method.

The log level of the publisher can be changed dynamically at any time using the `setLogLevel` method.

Server connections

Connecting to other Diffusion servers from within a publisher

Publishers can act as clients of other publishers for the purpose of distributed processing. In this case there is a client/server relationship between two publishers. One publisher can be a client of many other publishers and a publisher can have many publisher clients.

A publisher acting as the server in such a relationship sees the client as a normal client. The only thing distinguishing it is its client type (obtained using `Client.getClientType`).

However, for a publisher to act as the client of another publisher it must make an outbound connection to the Diffusion server that hosts the server publisher. In fact, the client publisher knows nothing of the server publisher, only the server and the topics it subscribes to, as if it were a normal client.

Server connections can be made automatically for a publisher by declaring them in `etc/Publishers.xml`. In this case the connections are made automatically during publisher startup. You can configure the behavior when such connections fail. It might cause the publisher to fail (if it is dependent upon the server as a data source) or it might allow the publisher to start but retry the connection periodically until it succeeds. Alternatively, it might do nothing.

Alternatively, the publisher can dynamically make server connections as, and when required. To do this it uses the `addServerConnection` method to create a `PublisherServerConnection` object, configure the connection as required and use the connect method on the object to make the connection

Whether server connections are made automatically or manually the publisher is notified when a connection is made using the `serverConnected` method and when the connection is closed or lost using the serverDisconnected method.

The publisher receives messages from server connections on the `messageFromServer` notification method.

The publisher is notified of the change of status (for example, removal) of any topics it is subscribed to at a server connection on the `serverTopicStatusChanged` notification method.

General utilities

General purpose utilities that can be used from within a publisher

There are a number of general purpose utilities available which can aid in the process of writing a publisher, for example:

Table 13: General publisher utilities

Utils	A set of general purpose utilities which include file handling, property handling, date and time formatting and more.
XMLUtils	A set of utilities to aid in the processing of XML.
HTTPUtils	A set of utilities to aid in HTTP processing.

Writing a publisher

How to approach writing a publisher

Publisher basics discusses the general concepts associated with publishers. This section goes into a little detail on what actually needs to be written in Java terms.

> **Note:** This section covers only the main aspects of the publisher API. See the API documentation for full details.

There are demo publishers issued with Diffusion which have the source provided and these act as examples of working publishers.

In its simplest sense a publisher is responsible for providing topics, and publishing messages relating to those topics.

Before a publisher is written you need to carefully consider what it needs to do and what methods need to be implemented. The areas that need to be considered and the methods relating to them are discussed below.

There are many ways to approach the design of a publisher. See design patterns for some possibilities.

Creating a Publisher class

A publisher is written by extending the abstract `Publisher` class (see Publisher API) and overriding any methods that must be implemented to achieve the functionality required by the publisher.

In all but the simplest of publishers it is likely that other classes must be written to perform the functionality required of the publisher.

The considerations of which methods must be overridden are discussed further within this section.

After the class is written and compiled, you can deploy it in the Diffusion server by specifying its details in `etc/Publishers.xml`

Publishers can also be deployed as a DAR file, sidestepping `etc/Publishers.xml`

See the section on testing for information about how to test the publisher.

Publisher startup

When a publisher is first loaded by the Diffusion server it can also be automatically started.

If not automatically started (or if it has been manually stopped), a publisher can be manually started by using the System Management interface. In either case the publisher processing goes through a number of startup steps. During these steps the `initialLoad` and `publisherStarted` methods are called and these methods can be implemented by the publisher to perform any initial processing like setting up the initial data state or adding initial topics.

Data state

A publisher typically holds some data which it updates according to any data update events it might receive.

The data held by the publisher is referred to as its state. To be more specific, it is the topics provided by the publisher that have data state. It is up to the publisher whether the data state is managed as a whole or on topic by topic basis.

It is the responsibility of the publisher to initialize its state and keep it updated as appropriate. Clients that subscribe to topics usually want to know the current state of the data relating to that topic and the publisher provides this as an initial topic load message. Clients are notified of all changes to that state by the publisher sending out delta messages.

A publisher typically has its own data model represented by classes written to support the data for the publisher. Ways in which such a data model can be managed are discussed in Data models.

Initial state

A publisher's data typically has some initial state which can be updated during the life of the publisher. The state clearly must be set up before a client requires it but exactly when this is done is up to the publisher.

The state of the data as a whole can be set up when the publisher starts. This can be done in the `initialLoad` method where all topics required can be set up and the data loaded as appropriate.

Alternatively, the state of the data relating to a topic can be initialized when the topic is added, which is not necessarily when the publisher is started.

Another option is that the initial state is provided by a data feed as it connects (or is connected to). When input is provided by an event publisher, the initial state can be set up when the event publisher connects or the event publisher can send an initial topic load message as its first message to the publisher. Similarly, if data is provided by a server connection, the initial state can be set up when the server connection is notified to the publisher or more typically the server provides an initial topic load message.

Data integrity

The integrity of the data is also the responsibility of the publisher and care must be taken to ensure that all updating of data state is thread-safe. For example, it must be borne in mind that a client can request a load of current state (for example, by subscription) at the same time as the state is being updated (for example, by `messageFromEventPublisher`).

> **Note:** The topic data feature automatically handles such data locking and in other cases topics might be locked as and when required.

Providing data state

If clients are to use the fetch facility to obtain the current state of topics, it will be necessary to consider the implementation of either the `fetchForClient` method of the publisher or a `TopicFetchHandler`.

Data inputs

For a publisher to be able to publish data to clients it must have a source for that data.

The data can be obtained from some type of feed, perhaps provided by some external API or it can be from some other application communicating using Diffusion protocols. This is entirely up to the publisher but Diffusion does offer some mechanisms.

Event publishers

Diffusion provides a mechanism for feeding data to publishers using the event publisher interface. The event publisher itself can also be written in Java using the event publisher API or it can be written in any other language using the event publisher protocol.

If a publisher is to receive messages from an event publisher, it must implement the `messageFromEventPublisher` method to process such messages. This transforms the data as required and update its own data state. Alternatively, it can specify topic listeners to handle the messages on a per topic basis.

For a publisher to know when an event publisher connection is accepted by its server or its connection is closed or lost, it must add an EventConnectionListener using `Publishers.addEventListener`. The publisher itself can implement `EventConnectionListener` or can delegate to another class. The listener is informed of connection through the `eventConnectionAccepted` method and of disconnection via `eventConnectionClosed`. These methods pass a reference to the EventConnection so that the publisher can determine to which event publisher it relates.

> **Note:** An event publisher might have connected before the publisher starts and is not a reliable source of initial state. You must consider how a publisher establishes its initial state in conjunction with an event publisher.

Server connections

A publisher can receive its data from another publisher in a distributed environment as if it were a client of that publisher. To do this it is necessary for the publisher to connect to the server of the remote publisher and subscribe to its topics.

Connection to remote servers can be set up automatically for a publisher by configuring the connection in `etc/Publishers.xml`. A reference to such a server connection can be obtained by the publisher using the `getServerConnection` method or all connections van be obtained using `getServerConnections`.

Alternatively, a publisher can explicitly connect to remote servers using `addServerConnection`.

However the connection is made, the publisher is notified of the connection via the `serverConnected` method.

If a server connection is closed by the remote server or lost the publisher is notified through the `serverDisconnected` method.

Once a serverConnection is established the publisher must subscribe to topics on it and receive topic messages through the `messageFromServer` method. Typically, the first such method is a load message providing the initial state for the topic. Processing of the messages can be done within this method or alternatively the publisher can specify topic listeners to handle the messages on a per topic basis.

See Publisher clients for more information.

Control clients

A publisher can receive input from a control client.

Control clients can use the TopicUpdateControl feature to publish messages to topics. Where such topics have topic data the topic state is automatically updated and deltas are published to subscribed clients. Where topics do not have topic data, published messages are forwarded to subscribed clients (that is, it is assumed that the control client maintains the data state).

Control clients can also send messages to specific clients and these are forwarded to the clients automatically.

Handling client subscriptions

Clients subscribe to topics provided by publishers and whenever this occurs the publisher is notified through its subscription method. The publisher can perform any processing it requires on subscription.

Topic loading

Typically, on subscription, the publisher provides the client with the current state of the data for the topic. It can do this by creating a new topic load message and populating it with a representation of the state. Rather than doing this every time a client subscribes it is generally more efficient for the publisher to create a topic load message only when the state changes and send this same message out to every client that subscribes.

This provision of the current state is known as the topic load. This can be done in one of the following ways:

Topic load in `subscription` method

If the topic has not already been loaded by a topic loader (see below), the loaded parameter of the subscription method is false. In this case, the normal action is for the publisher to send a topic load message to the client (passed as a parameter to subscription) through its send method.

Topic loaders

A topic loader is an object that implements the `TopicLoader` interface and can be used to perform any topic load processing that is required for one or more topics. Topic loaders can be declared for a `Publisher` using the `Publisher.addTopicLoader` method. This is typically done in the `initialLoad` processing and must be done before any topics that are loaded by the topic loader are added.

Using topic data

The use of topic data is recommended for topic data management and if it is in use then topic loading is fairly automatic and the default `subscription` method deals with it.

Hierarchic subscription

When a client subscribes to a topic the publisher can choose to subscribe the client to other topics or to subordinate topics. This can be done using the `Client.subscribe` methods.

A client itself can request subscription to a hierarchy of topics using topic selectors but this is an alternative method of handling hierarchies.

Publishing messages

Publishing a message means sending it to all clients subscribed to a topic. The message itself nominates the topic to which it relates.

The most common reason for publishing a message is to send an update (or delta) to the topic's data to all subscribed clients but messages can be published for any reason. In the case of an update, this is done in the method that became aware of the update (for example, `messageFromEventPublisher`, `messageFromServer` etc).

A message for publishing can be created and populated by the publisher and then published using publishing methods on the topic or the publisher itself.

Exclusive messages

To send a message from a publisher to all clients subscribed to a topic except one single client, it can use the publishExclusiveMessage method. This might be appropriate if the message being published is a result of receiving a message from a client which you do not want to send back to that client.

Message priority

The priority at which a message is to be sent can be altered from the normal priority. For example, an urgent message can be sent with high priority causing it to go to the front of the client's queue.

Message acknowledgment

If acknowledgment of a message is required then it can be set as an acknowledged message. If any clients do not respond to a acknowledged message within a specified timeout, the publisher is notified on its `messageNotAcknowledged` method.

Topic locking

When TopicData is in use, all locking of the topic state is handled automatically. However, when the state of the topic is maintained in some other manner (for example as a topic attachment), it is the responsibility of the publisher application to handle locking.

About this task

The publisher must consider the issue of locking the topic whilst its state is changed and delta messages published.

By default, all topics have locking enabled which allows the publisher to lock and unlock the topic as required. When a client subscribes to a topic the subscription method of the publisher, normally sends the current state of the topic to the client. With locking enabled, the topic is locked for the duration of the subscription, which prevents other threads from acquiring the topic lock.

Threads that update the topic state and publish messages must also lock the topic for the duration of the update and publish. If this technique is not employed, a delta message might be sent to a client before the subscription method has the opportunity to send a topic load message. This can cause a failure at the client if topic aliasing is in use as aliasing relies upon the topic load message reaching the client first. Even if topic aliasing were not in use, the client application must be prepared for a delta arriving before the topic load.

Procedure

The following code example shows how a method responsible for updating the topic state and publishing a delta handles topic locking:

Results

```
void updateMyData(String newValue) throws APIException {
    theTopic.lock();
    try {
        MyData data = (MyData)theTopic.attachment();
```

```
            data.setValue(newValue);
            TopicMessage message = theTopic.createDeltaMessage();
            message.put(newValue);
            publishMessage(message);
        }
        finally {
            theTopic.unlock();
        }
    }
```

Using the try/finally technique ensures that whatever happens the code that locks the topic also unlocks it, which prevents a lock being left on indefinitely. As mentioned earlier the topic is automatically locked during subscription so the above code prevents an update from occurring whilst a subscription is in progress.

Handling clients

Interacting with clients from within a publisher

A publisher is notified when a client subscribes to one of its topics through the `subscription` method and when the client unsubscribes the `unsubscription` method is called.

A publisher can receive message from clients and send messages to clients (see below).

A client can request the state of any topic or topics at any time even if not subscribed to it. This is referred to as 'fetch' request. Such a request can routed to the publisher's `fetchForClient` method if a topic has no `TopicFetchHandler` declared and does not use topicData.

Other than the above, a publisher is not normally notified of any other client activity. However a publisher can choose to receive client notifications using the `Publishers.addEventListener` method. Using client notifications, a publisher can even handle a fetch request for a topic that does not exist and return a reply (using `Client.sendFetchReply`) without the overhead of actually creating a topic.

A publisher can also choose to close or abort clients.

Sending and receiving client messages

In addition to publishing messages to all clients subscribed to a topic, you can send a message to only a single client using the `Client.send` method.

A client can also send messages to the publisher and these are received on the `messageFromClient` method which handles them accordingly. Only implement this method if messages are expected from clients. Alternatively the publisher specifies topic listeners to handle the messages on a per topic basis.

The 'client groups' facility allows messages to be sent to all clients in a group. There is also an 'exclusive message' facility which caters for sending to all but one client in a group.

Publisher closedown

A publisher is stopped and removed when the Diffusion server closes but can also be stopped and restarted, or stopped and removed by using the System Management interface.

However a publisher is stopped it always goes through a set of closedown steps, during which the `publisherStopping` and `publisherStopped` methods are called. A publisher can implement these methods if required to perform any special processing such as tidying up resources used.

Publisher removal

When a publisher is finally removed (either during server closedown or by using System Management), it cannot be restarted again within the same server instance. After removal the publisherRemoved method is called and this gives the publisher the opportunity to perform any final tidy up processing.

Stopping and restarting using System Management

By default, you cannot stop and restart a publisher using the System Management functions because in order for this to work the publisher must cater for the integrity of its state when this happens. As topics are also removed during stopping, the publisher must also be able to restore these topics if it were restarted.

If a publisher does want to cater for stop and restart using System Management, it must override the `isStoppable` method to return true. The publisher code must be able to recover topics and data state on restart.

Testing a publisher

There are various ways you can test your publishers after you have written them and deployed them on a Diffusion server instance.

The easiest way to perform some initial tests is to start it and try it out using some of the supplied testing tools. For example, you can start one or more instances of the client test tool, connect each to the test server and subscribe to the publisher's topic or topics. The initial topic load data is displayed and any messages sent as deltas are also displayed in each client. This tool can also be used to send messages to the publisher from the client.

You can simulate data being sent from an event publisher using the event publisher test tool.

Ultimately such tests are limited and you might want to develop Java tests which simulate clients using the Java external client API (or the Windows external client API).

Test as soon as possible with the actual clients that are going to be used. So, for example, you might want to develop browser clients using HTTP, Flash or Silverlight.

It can help to diagnose problems with the publisher if it has diagnostic logging encoded within it. Such logging can be provided only at fine level and this logging level used only during testing.

Client queues

How messages sent to clients are queued and how such queues can be manipulated by publishers

The Diffusion server maintains an outbound queue of messages for each client. Whenever a message is published to a topic, it is placed in the queue of each client subscribed to that topic as will any message sent explicitly to the client. Messages are sent to the client strictly in the order that they are enqueued.

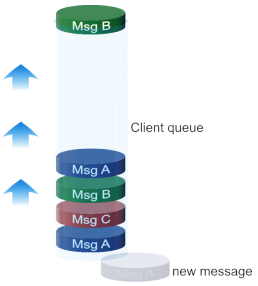


Figure 11: The message queue

A publisher is able to enquire upon the details of a particular client's queue and even to change some aspects of the queue's behavior.

Queue enquiries

How the publisher can access details of client queues

A publisher can enquire upon the following information about a particular client's queue using the client interface:

- The current queue size
- The maximum queue size (The limit the queue can reach before the client is automatically disconnected.)
- The largest queue size (The largest size the client queue has been since the client connected.)

Maximum queue depth

To limit the backlog of messages queued for a client that is not consuming them quickly enough you can indicate a maximum queue depth for clients.

Choose this size carefully as a large queue size can lead to excessive memory usage and vulnerability to Denial of Service attacks, whilst a small queue size can lead to slow clients being disconnected too frequently.

The maximum queue depth for clients can be configured for a client connector in `etc/Connectors.xml`. A default value can also be configured in `etc/Server.xml` for connectors that do not explicitly specify a value.

These values can be changed dynamically at run time using System Management but they only take effect for new clients.

Queue notification thresholds

A publisher can receive notifications when a client queue has reached a certain size and use this information to decide whether or not to act on the situation.

For example, the publisher might want to notify the client so that it can take some action (like suspending processing). As there is little point in queuing a message to tell the client that their queue is becoming full, this is probably done using a high priority message which goes to the front of the queue.

To this end, an upper notification threshold can be set for a client's queue. This is expressed as a percentage of the maximum queue depth at which a notification is sent to any client listeners that are declared. A client listener is any object that implements the `ClientListener` interface and such a listener can be added from within a publisher using the `Publishers.addEventListener` method. Listeners are notified of the limit breach using the `clientQueueThresholdReached` method.

In addition a lower notification threshold can be specified. The lower threshold is a percentage of the maximum queue depth at which a notification occurs when a message is removed from the queue causing the queue size to reach the threshold if (and only if) the upper threshold has been breached.

When the `clientQueueThresholdReached` method is called on the client listener it indicates whether it was the upper or lower threshold that was reached.

The thresholds to use for clients can be configured for a connector in connectors.properties. If not specified, the default thresholds specified in `etc/Server.xml` are used.

The thresholds on a client connector can be changed dynamically at run time using System Management, but the new values only take effect for new clients.

Thresholds can also be set or changed from within the publisher for a connected client using the `Client.setQueueNotificationThresholds` method.

Tidy on unsubscribe

When a client unsubscribes from a topic, the topic updates that are already queued for delivery to the client are delivered. These messagers can be cleared from the queue if the client does not want to receive them.

After a message is queued for a client, its delivery is guaranteed. This means that a client can unsubscribe from a topic but still receive messages queued for it on it on that topic. This is generally what is required as the messages were sent whilst the client was subscribed.

However, it can be decided that once the client has unsubscribed from a topic then the client no longer has any interest in any messages for that topic and such messages are removed from the queue. To achieve this there is an option on a topic (using the `setTidyOnUnsubscribe` method) to indicate that messages for the topic must be removed from client queues when the client unsubscribes from that topic.

Filtering queued messages

Filtering the messages that get queued for a client

You can filter the messages that get queued for a particular client. For more information, see Message filters on page 214.

Client validation

Diffusion provides the facility to validate client connections to determine whether they are allowed using whitelists (allowed clients) and blacklists (barred clients). This can be made to occur automatically or can be controlled programmatically from within publishers.

Validation can occur at the following levels:

- On connection (When a client connects to Diffusion you can validate whether the client is allowed to connect (regardless of topic being requested).)
- On subscription (When a client subscribes to a topic you can validate whether the client is allowed to subscribe to the topic.)

You can allow/disallow clients explicitly using their IP address (or host name) or by geographical criteria determined by client Resolution (for example, you can bar all clients from a particular country). You can also perform fuzzy matching on address or geographical values (for example, you can bar all clients with a host name that starts with a specified string).

The specification of the clients that are to be allowed/disallowed is known as a client validation policy. There can be one client connection validation policy for a Diffusion server and there can also be a client subscription validation policy specified for each publisher.

The policies are maintained as files but can be updated programmatically from within publisher code. You can dynamically update a policy during publisher processing.

The format of the XML policy files is defined by the issued XSDs and is also defined in the configuration section.

Client validation policy types

A client validation policy can be a whitelist or a blacklist.

With a whitelist only those clients who match the criteria specified within the policy are allowed to connect or subscribe, whereas with a blacklist any clients who match the criteria specified within the policy is prevented from connecting or subscribing.

Automatic or manual policies

You can declare a client validation policy as automatic, in which case the policy is applied by the server, or manual, in which case the policy is applied by a publisher.

If a client validation policy is declared as being *automatic*, it is applied automatically by Diffusion and clients that do not pass the client validation are automatically disconnected (for a connection policy) or unsubscribed (for a subscription policy).

A policy can instead be defined as *manual* in which case Diffusion does not automatically apply the policy. In this case the policy can be manually applied using the publisher API and the publisher can decide what action to take, not necessarily disconnect or unsubscribe.

Client validation criteria

The criteria by which a client is validated as being allowed or disallowed is specified in terms of selection values.

The value types are as follows:

Table 14: Validation criteria

address	The client's IP address
host name	The client's host name
resolved name	The client's resolved name
country	The client's country code
language	The client's language code

The values (other than address) are as resolved by the Diffusion WhoIs service and the WhoIs service must be enabled if used. When resolved values are used, the comparison cannot be made until the client is resolved. This can mean that a client that has not yet been resolved might be allowed to connect but is then disconnected when it is resolved and found to be disallowed by one of the resolved policy criteria.

For each type the policy can specify any number of values to check against. Each value can be an absolute value (for example, an IP address) or a regular expression pattern which is evaluated against the actual client value.

When a client connects or subscribes, each of its actual values are checked against the policy lists for a match.

A blacklist policy disallows any clients that have values matching a value within any value type specified within the policy. Value types can be mixed. For example, you can specify a country code and a list of excluded IP addresses which causes all clients from the specified countries to be excluded plus those with the specified addresses.

A whitelist policy allows only clients that match at least one value from each of the specified value types. For this reason it does not usually make sense for more than one value type to be specified in a whitelist.

For example, consider the following value lists:

```
address=194.124.6.78 & 195.233.3.4
hostName=XYZ*
```

If this is a blacklist, it disallows the IP addresses specified plus any client that has a resolved host name starting with “XYZ”. However, if this is a whitelist it allows only a client if it had one of the specified IP addresses and a host name starting with “XYZ”.

Client connection validation policy

The client connection validation policy applies to clients connecting to a Diffusion server. Each connector can have its own policy file, which is specified in `etc/Connectors.xml`.

If an automatic policy has only address information (that is, no criteria that require client resolution), any client disallowed by the policy is not allowed to connect. If the policy does have criteria that require client resolution and the client connecting has not been resolved, it is allowed to connect but can be disconnected as soon as the client is resolved.

Client subscription validation policy

A client subscription validation policy applies to clients subscribing to topics belonging to the publisher that the policy is specified for. Each publisher can have its own policy file, which is specified in `etc/Publishers.xml`

If an automatic policy has only address information (that is, no criteria that require client resolution), any client disallowed by the policy is not allowed to subscribe to any topic indicated by the policy. If the policy does have criteria that require client resolution and the client subscribing has not been resolved, it is allowed to subscribe but can be unsubscribed as soon as the client is resolved.

Using policies programmatically

Connection and subscription validation policies can be defined and changed at runtime using the configuration interface. Even if the policies were loaded from XML they can be changed at runtime, even after the server has started.

Connection validation policies can added to connectors or existing policies defined for connectors can be amended.

Subscription validation policies can be defined for publishers or existing policies can be amended.

Client Geo and WhoIs information

When a client connects to Diffusion, information about that client's geographic location is looked up and the information is made available to publishers.

When a client first connects to a Diffusion server, its remote IP address is immediately available (using the `Client.getRemoteAddress` method) as well as other details obtained from the embedded GeoIp database. Further host and geographic details about the client are obtained using the Diffusion "`WhoIs` service".

GeoIp information

Diffusion ships with a GeoIP database from MaxMind. This provides information about Locale and geographic co-ordinates. The Java API includes utilities (`GeoIPUtils`) to make use of this database.

This is a public domain database and is free to use. You can purchase the more accurate database from MaxMind and change the configuration in the `etc/Server.xml` properties to use the new database.

The database can be disabled but its use is mandatory if you are going to use client connection or subscription validation policies.

WhoIs

The inbuilt WhoIs service can provide additional information about clients, however the lookup of the WhoIs information might take some time, especially if it is not already cached. This means that notification of the connection and further processing of the client cannot wait for this information to become available. For this reason the resolution of the client's WhoIs details is notified to client listeners separately from client connection on the `clientResolved` method.

When a client is first connected it is likely that the WhoIs details of the client are not available. This can be checked using the `Client.isResolved` method. When the details become available they can be obtained from the client using the `getWhoIsDetails` method which returns an object containing the following information:

Table 15: WhoIs

Address	The client's IP Address – this is the same as that obtained using `Client.getRemoteAddress`.

Host	The resolved host name of the client. If the host name cannot be resolved, the address is returned.
Resolved name	The fully resolved name of the client. Exactly what this means depends upon the WhoIs provider in use. If a fully resolved name cannot be obtained, the host name value is returned.
Locale	Returns the result of a geographic lookup of the IP address indicating where the address was allocated. The country of the locale is set to the international two-letter code for the country where the internet address was allocated (for example, NZ for New Zealand). If the internet address cannot be found in the database, the country and language of the returned locale are set to empty Strings. Three country values can be returned that do not exist within the international standard (ISO 3166). These are EU (for a non-specific European address), AP (for a non-specific Asia-Pacific address) and ** (an internet address reserved for private use, for example on a corporate network not available from the public internet). The language of the returned locale is set to the international two-letter code for the official language of the country where the internet address was allocated. Where a country has more than one official language, the language is set to that which has the majority of native speakers. For example, the language for Canada is set to English (en) rather than French (fr). Non-specific addresses (EU and AP), private internet addresses (**), and addresses not found within the database, all return an empty string for language.
WhoIsData	This is data extracted from an enquiry upon a 'WhoIs' data provider.
Local	Indicates whether the client address is a local address, in which case no locale or WhoIsData is available.
Loopback	Indicates whether the client address is a loopback address in which case no locale or WhoIsData is available.

The Diffusion WhoIs service

The Diffusion WhoIs service runs as a background task in the Diffusion server. It looks up client details and caches them in case the same client reconnects later.

The behavior of the WhoIs service is configured in `etc/Server.xml`. This allows the following to be specified:

Table 16: WhoIs service

The WhoIs provider	This specifies a class to use for WhoIs lookups. A default WhoIs provider is provided with Diffusion.
Number of threads	The number of background threads that processes `WhoIs` resolver requests. More threads will improve the WhoIs performance. Setting this to 0 disables WhoIs.
WhoIs Host/Port	These details provide the location of an internet based WhoIs lookup server that adheres to the RFC3912 WhoIs protocol. This is used by the default WhoIs provider. This defaults to using the RIPE database.

Cache details	Specifying the maximum size of the cache of details and how long cache entries are retained before being deleted. If you envisage large numbers of different clients connecting over time, it is important to consider the automatic cache tidying options on the service.

The WhoIs service can be disabled both by setting the number of threads to zero and removing the whois configuration element.

WhoIs providers

The Diffusion WhoIs provider is a class which implements the `WhoIsProvider` interface of the `WhoIs` API. This is used by the `WhoIs` service to lookup WhoIs details for connected clients.

Default provider

A default `WhoIsProvider` (`WhoIsDefaultProvider`) is provided with Diffusion.

A connection is made to the WhoIs server specified in `etc/Server.xml` and returned details are parsed and used to update the supplied details. Child details objects are added for any separate WhoIs records found and the type of such objects is the key of the first WhoIs record entry (for example, "person"). Where duplicate field names occur then all but the first are suffixed by "*_n*", where *n* is a number distinguishing the entries.

The netname entry is used as the resolved name if present.

Custom provider

If the behavior of the issued default WhoIs provider is not exactly what is required then users can write their own WhoIs provider which must implement the `WhoIsProvider` interface. The name of the user-written class can be specified in `etc/Server.xml` and must be deployed on the Diffusion server's classpath.

Client groups

Clients that are connected can be managed in client groups allowing messages to be sent to groups of clients

Client groups are a convenient way of managing groups of clients with common attributes.

A publisher can create a client group and add clients to it. Messages can be sent to the group of clients rather than to individual clients.

When a client disconnects it is automatically removed from all client groups of which it is a member.

A client group belongs to the publisher that created it and can be used only from within that publisher. Group names are unique within the publisher only.

Client groups are a feature of the Java API using the `Publisher` interface.

Creating a client group

A publisher can create a new client group at any time using `createClientGroup`. When creating the group it is given a name which must be unique within the publisher. When a client group is created a reference to a `ClientGroup` object is returned.

Adding clients to a group

Clients can be added to a client group using the `addClient` method on a `ClientGroup` object.

Sending messages to clients in a group

A message can be sent to all clients in a client group using the `ClientGroup.send` method. To send to all but a specified client use `publishExclusiveMessage`.

Removing clients from a group

To remove clients from a client group, use the `ClientGroup.removeClient` method.

Removing a client group

To delete / remove a client group that is no longer required, use the `Publisher.removeClientGroup` method.

Other methods

All clients within a client group can be listed using `ClientGroup.getClients`.

You can check whether a particular client is already a member of a client group using `ClientGroup.containsClient`.

To enquire upon which client groups a particular client belongs to you can use `Publisher.getClientGroupMembership`.

To get a reference to a named client group from a publisher use `getClientGroup`.

Temporary client groups

Normally client groups have publisher scope, but you can create temporary groups using the `union` and `intersect` methods. These methods allow for the creation of client groups which are not managed by the client group manager. These temporary groups allow for the sending of messages which contain clients from different groups.

Client notifications

A publisher can opt to receive certain notifications regarding clients. It does this by adding a `ClientListener` which can be the publisher itself or any other object that implements the `ClientListener` interface.

A listener is added using the `Publishers.addEventListener` method.

All notifications are passed a reference to the client in question which can be interrogated for further information as required.

Notifications received on the `ClientListener` interface are as follows:

Table 17: Client listener notifications

`clientConnected`	This is called whenever a new client connects. It is not necessarily a client that is subscribing to one of the publisher's topics.
`clientResolved`	This is called when a newly connected client is resolved. A client's full geographical information is not necessarily available as soon as a client connects and so this method is called separately after the client has been resolved.

`clientSubscriptionInvalid`	This is called whenever a client attempts to subscribe to a topic that does not exist. This might be because the topic is not yet available and this gives a publisher the opportunity to create the topic and subscribe the client to it.
`clientFetchInvalid`	This is called whenever a client attempts to fetch a topic that does not exist. This gives the publisher the opportunity to respond to fetch request on a non-existent topic. A publisher can even reply to such a request without having to create a topic using the `sendFetchReply` method.
`clientSendInvalid`	This is called whenever a client attempts to send a message to a topic that does not exist, or to which the client is not subscribed. This enables a client to send a message to a topic and for that topic to be created and subscribed to on demand, or send data when a response is never expected.
`clientQueueThresholdReached`	This is called whenever a client's queue breaches an upper queue notification threshold or returns to a lower queue notification threshold. Parameters indicate which threshold has been reached and the threshold value.
`clientCredentials`	This is called whenever a client supplies new credentials after connection. It is called after the authentication handlers and authorization handlers (if any exist) have validated the credentials.
`clientClosed`	This is called whenever a client disconnects. The reason for disconnection can be obtained using `theClient.getCloseReason` method.

Adding a ClientListener

You can add a `ClientListener` to listen for client notifications.

About this task

Procedure

So a publisher can add itself as a listener for client notifications as follows:

Results

```
public class MyPublisher extends Publisher implements ClientListener {

    protected void initialLoad() throws APIException {
        Publishers.addEventListener(this);
    }
```

Using DefaultClientListener

How to use the default client listener to avoid implementing all methods.

About this task

The publisher must implement all of the `ClientListener` methods.

Procedure

For convenience, an abstract `DefaultClientListener` class is provided which has empty implementations of all methods. This can be extended to produce a class which implements only the methods you are interested in. Alternatively an anonymous class can be used within the publisher as follows:

Results

```
    protected void initialLoad() throws APIException {
        Publishers.addEventListener(
            new DefaultClientListener() {
                public void clientConnected(Client client) {
                    LOG.info("Client {} connected",client);
                }

                public void clientClosed(Client client) {
                    LOG.info("Client {} closed",client);
                }
            });
    }
```

Design patterns

There are many ways in which you can approach designing and writing a publisher. These ways can be significantly different from application to application.

This section provides some hints and tips and common patterns that might be employed when writing publishers.

Data models

Almost all publishers must manage data in some way. It is necessary to design the data model that holds the data within the publisher.

This needs to take into consideration the format of the data that comes into the publisher and how it is transformed into the publisher data. More importantly it needs to consider the format of the data that is sent on each topic to clients in the form of initial load and delta messages.

When data is fed to a publisher through an event publisher or server connection, it is likely that the data is already transformed into a suitable format for the publisher by the sending application, but further transformation can still be required before the data is in a suitable format for the publisher's topics. Data from other sources must be transformed into formats suitable for the topics. These data transformations represent a large part of what a publisher does.

Topic data

A useful pattern for handling data within a publisher is by associating the data with the topics themselves.

Topics have a feature called topic data which greatly simplifies the handling of data associated with a topic. A number of different data types are catered for as well as the ability to write a custom data handler. Topic data provides mechanisms for data locking, data state management and message caching. It also provides mechanisms for automatically generating deltas by comparing incoming data updates with the current data state.

The topic class also has the ability to attach an object of any type. This mechanism can also be used to associate some type of data handling object with the topic when the use of topic data is not appropriate.

Topic structure

Within Diffusion topics are organized into a topic tree. This provides for a high level of control over the organization of topics and how they are handled.

For example, clients can subscribe to branches of the tree instead of all of the individual topics

Of course, a publisher might be handling only a very small number of topics or it might be inappropriate to organize the topics in a tree, in which case a flat structure can be used.

Flat topic organization

In some cases a flat topic structure might be appropriate. Reasons for this can be as follows:

- There are very few topics.
- The topics have no logical organization.
- Topic name lengths are to be kept to a minimum (although the use of topic aliasing means there is little reason to do this).

In this case topics are added individually within the publisher.

Topic tree organization

A tree of topics can be employed when there is a need for more flexibility of topic control. Where data is logically hierarchical in nature anyway, it can be very advantageous to arrange the topics to match the data for the following reasons:

- Fine-grain data subscription. (Clients subscribe to only the individual topics they want data for, minimizing the data that is sent to the client. Also, when data changes, only the data for the topic is sent and not all data.)
- Subscription Control. (Clients can subscribe to branches of the topic tree without subscribing to individual topics.)

> **Note:** When longer topic names are in use, the effect on messages sizes can be considerably reduced by using the topic aliasing feature.

Related Links

Topic tree on page 130
Topics are organized in a tree structure. Paths to topics consist of level names separated by the slash character (/). A topic can be bound to any level of the hierarchy.

Chapter 5

Clients

In this section:

- Connection
- Topics
- Messages
- Client types
- Client close reason
- Cross domain
- Protocols supported
- Browsers supported
- Browser limitations

In Diffusion terms a client is any application that connects to a Diffusion server using a Diffusion client API.

You can implement clients that have the following roles in your Diffusion architecture:

- Standard clients that have only non-control capabilities, such as subscribing to topics or sending messages to topics. These kinds of clients can be implemented in either the Classic API or the Unified API
- Control clients that can control or handle events on the server, such as creating and publishing to topics or authenticating other clients. These kinds of clients can be implemented only in the Unified API

This section of the manual includes information that applies to both of these client roles. For information that is specific to control clients. For more information, see Control client on page 226.

Connection

A client must connect to a Diffusion server to communicate with Diffusion.

Connections can be HTTP or through a TCP/IP socket connection depending upon Client type.

A TCP client must specify the host and port to connect to. The port corresponds to the port that a connector of the type corresponding to the client type is listening on at a Diffusion server on the specified host machine.

If the server is configured to support client reconnection and the client API also supports it, a client disconnected due to IO error can attempt to reconnect without losing topic state or any messages queued for the client whilst disconnected.

For more information about the specifics of client connection, see the API documentation.

Connection details in the Classic API

Clients can be configured to failover to, load balance between, or cascade on connection to a list of pre-defined servers.

A client can accept a `ConnectionDetails` object on construction, which contains a list of one or more `ServerDetails` objects.

A `ServerDetails` object contains:

- **URL** – The URL specifying protocol, host, and port.
- **Connection timeout** – The socket connection timeout in milliseconds.
- **TopicSet** – The TopicSet for this connection.
- **Credentials** – Client credentials for this connection.

If any of the `ServerDetails` attributes are missing, the ConnectionDetails attributes are used, if set.

The `ServerDetails` can be used in any combination of the following three ways:

Auto Failover	If a successful connection to the a server is made which later fails, the next `ServerDetails` object in the list is used to attempt another connection. If there are no more `ServerDetails` to choose from, the client is sent a closed connection notification.
Load Balance	The server details list is shuffled prior to use to ensure that not every client connects to the same server initially. If auto failover is enabled, these clients failover in the same way as above, but each to randomly ordered lists of `ServerDetails`.
Cascade	Similar to auto failover, but with the crucial logic taking place prior to a connection. If a client attempts connection to a server and fails, the next `ServerDetails` object is chosen and the connection is attempted on the new server.

These behaviors can be enabled with the relevant setter methods of a ConnectionDetails object.

Connection Details Listener

An optional `ConnectionDetailsListener` receives events from Diffusion Clients as the events described above take place. The callbacks received are:

ConnectionDetailsAcquired

Prior to a connection attempt being made, the ServerDetails object is passed as an argument to the `ConnectionDetailsAcquired` method/event handler.

ConnectionSequenceExhausted

Called when the list of ServerDetails is exhausted. No arguments are supplied.

Connecting through an HTTP proxy

Clients can connect to the Diffusion server through an HTTP proxy by using the HTTP `CONNECT` verb to create the connection and tunneling any of the supported transports through that connection.

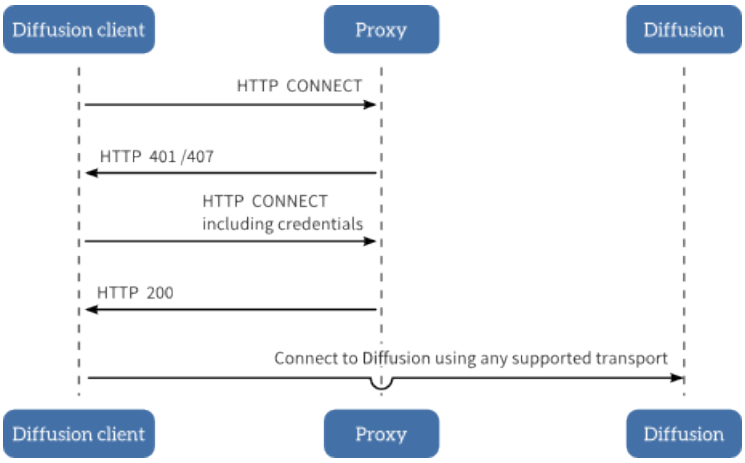


Figure 12: Flow of requests and responses when connecting to Diffusion through a proxy.

In the Classic API

Clients that use the Classic API to connect to the Diffusion server, can connect through a proxy only if the proxy does not require authentication.

In the Unified API

Java clients that use the Unified API to connect to the Diffusion server, can connect through an HTTP proxy under the following circumstances:

- If the proxy does not require authentication.

  When creating your session, add an HTTP proxy to the session by passing in the host and port number of the proxy.

  ```
  Diffusion.sessions().httpProxy(host, port)
  ```

- If the proxy requires basic authentication, the client can use the implementation in the Unified API to authenticate.

  When creating your session, add an HTTP proxy to the session by passing in the host and port number of the proxy and a proxy authentication object that provides the challenge handler for basic authentication.

  ```
  HTTPProxyAuthentication auth =
   Diffusion.proxyAuthentication().basic(username, password);
  Diffusion.sessions().httpProxy(host, port, auth);
  ```

- If the proxy requires another form of authentication, you can implement a challenge handler that the client uses to authenticate.

  Implement the `HTTPProxyAuthentication` interface to provide a challenge handler that can handle the type of authentication your proxy uses. When creating your session, add an HTTP proxy to the session by passing in the host and port number of the proxy and a proxy authentication object that provides your challenge handler.

> **Note:** The proxy authentication mechanism is separate from the client authentication mechanism and is transparent to the Diffusion server.

Topics

A client can subscribe to topics and receive messages on all topics to which it subscribes.

Topics for subscription to can optionally be supplied on connection and further topic subscriptions can be requested at any time during the connection.

A client can subscribe to any number of topics.

Topics can be specified either by providing explicit Topic names or by using Topic selectors.

A client can also |fetch the current state of a topic without being subscribed to it. This effectively provides an asynchronous request/response mechanism. Headers sent with the fetch request are reflected back with the reply allowing for correlation of requests and responses. A client can request a fetch for a topic that does not exist as long as the publisher handles it and provides request/response without topics.

Messages

Typically, after subscription a client is sent a topic load message containing the current topic state and from then on, until unsubscription or disconnection, receives updates to the topic in the form of delta messages.

Clients can also receive messages sent directly to them from publishers and can also send messages to publishers (through their topics). The client can indicate that messages sent to the publisher require acknowledgment.

Clients can ping the server with a Ping message which causes the server to send a response from which the latency of the connection can be established.

Client types

There are many different types of client that can connect to Diffusion

The exact functionality available to a client application depends upon the client type and the Diffusion API used (if any).

There are many types of client as follows:

Table 18: Client types

External	This is a broadly generic term for any client that connects to Diffusion using DPT. This can be an application using the Java client API or the .NET client API. It can also be any other application that communicates directly using DPT. Full duplex connections over HTTP are catered for by the external client APIs.
JavaScript	This is a client that connects from a web browser using the Diffusion JavaScript client APIs which allows JavaScript to encapsulate a plugin client (.NET or Silverlight) or use a generic transports like iFrame, Callback Polling HTTP or WebSockets.
Flash	This is a client that connects from a web browser using the Diffusion ActionScript API which enables web applications to exploit the Adobe® Flash platform.

Silverlight	This is a client that connects from a web browser using the Diffusion Silverlight API which enables web applications to exploit the Microsoft® Silverlight framework.
WebSockets	This is a client that connects from a web browser that supports WebSockets or the Java, .NET or iOS client implementation.
Publisher	A Diffusion publisher can be a client of another Diffusion publisher. It is this facility that provides distribution within Diffusion.
Android	This is a client that connects from an Android phone using the Android client library, which uses the Diffusion Android phone API.
iOS	This is a client that connects from an iOS device using the Objective C library, which uses the Diffusion iOS API.

External clients

An external client is an application that connects to Diffusion using DPT.

An external client can use the Java client API or the .NET client API or can be written in any language that can read and/write over TCP sockets.

API external clients can connect using TCP or HTTP or other Diffusion protocols. The transports provided vary by client but the Java client provides all transports except the HTTP Comet-based transport that is supported only in Flash.

For more information about which clients support which protocols, see Protocols supported on page 115

Secure connections

External clients using the client APIs can also connect over a secure (SSL) connection. Again available secure connections vary between clients. The .NET external client does not provide HTTPS, it does allow secure DPT connections to be made. If you require external client that supports HTTPS, you can use the Java external client.

Proxy connections

External clients using the client APIs can connect through an HTTP proxy.

When using a client-side proxy, do not enable connection pooling or connection sharing for Diffusion. Ensure that the proxy does not route requests from more than one client through the same TCP connection.

If more than one client uses the same TCP connection, the Diffusion server cannot tell which client owns the connection. This can cause problems. For example, if the Diffusion server closes the shared TCP connection for reasons that are related to a single client that uses the connection, the proxy can close its connections to all clients that used the shared TCP connection.

JavaScript clients

JavaScript clients use a JavaScript library called `diffusion.js` to connect to and interact with Diffusion from within a supported web browser.

For information about which browsers are supported for a JavaScript client, see Browsers supported on page 118.

The transport that they use to connect to the Diffusion server has been encapsulated in the DiffusionClient singleton. To connect to a Diffusion server, pass a connection object into the connect method of DiffusionClient. The full list of parameters are provided in the jsdoc

Liveness monitor

The JavaScript API implements a liveness monitor that listens for pings from the server and raises an event if the connection to the server is lost. For more information, see Reconnecting with the JavaScript API on page 357.

Silverlight clients

A Silverlight client uses the `PushTechnology.Transports.dll` assembly (found in the `clients/silverlight` directory of a Diffusion installation).

API Silverlight clients can connect using DPT, or HTTP; they can also be configured to cascade. This means that the clients try the socket connection first, and if that fails or is blocked, an HTTP connection is attempted.

This client can also connect over a secure (SSL) connection using HTTPS

Port selection

When configuring Diffusion, it is worth thinking about which port to use with Silverlight policy file. DPT uses port 943. HTTP can use any port. Bear in mind the security implications of this behavior.

Security implications

As of Silverlight 2 onwards, the security rules have changed. Silverlight tries to get a policy file from port 943, if it cannot get a policy file from this port, the connection is refused. Diffusion understands these requests so it is important that you have supplied the correct policy file to the Silverlight connector and the policy file connector. An example of a policy file is included in the etc directory, although this is configured to be very open.

Flash clients

A Flash client uses `diffusion-flex.swc` (found in the `clients/flex` directory of a Diffusion install).

Flash clients can connect using DPT, HTTP or HTTPC. They can also be configured to cascade. This means that the client tries the socket connection first, and if that fails or is blocked, tries a HTTP connection.

This client can also connect over a secure (SSL) connection using HTTPCS or HTTPS

Port selection

When configuring Diffusion, it is worth thinking about what port to use for the Flash socket. Any port can be used, but you must bear in mind the security implications. Clients have had good success having the port set to 443, as this port is open in most firewall situations.

Security implications

As of Flash Player version 9, the security rules have changed. Flash tries and get a policy file from port 843, if it cannot get a policy file from port 843, it uses the port configured for the connection to get a policy file. Diffusion understands these requests so it is important that you have supplied the correct policy file to the Flash connector and the policy file connector. An example of a policy file is included in the `etc` directory, although this is configured to be very open.

To use HTTP or HTTPC a cross-domain policy has to be configured. Please refer to the Cross-domain section for more information.

DPTS connections

If your client uses Flash Player version 10 (or later), it can connect using DPTS. To connect using DPTS, the client application must be loaded inside an SSL page and must attempt to connect to a port that has SSL enabled.

The client attempts to get a policy file from port 843 (see Security implications on page 109). To serve a policy file for a DPTS connection, port 843 must have SSL enabled.

Enable SSL on a port by adding a key-store to the `Connectors.xml` file entry for the port. For more information, see Network security on page 432 and Connectors.

Liveness monitor

The ActionScript API implements a liveness monitor that listens for pings from the server and raises an event if the connection to the server is lost. For more information, see Reconnecting with the ActionScript API on page 363.

WebSocket clients

A client can connect over the WebSocket protocol

WebSocket clients can be used with the JavaScript API, .NET API, iOS API, or the Java API.

WebSocket clients use the TLS / SSL protocol if given the wss: schema.

Publisher clients

A publisher client refers to a publisher which is connected to a Diffusion server as if it were a client for the purpose of distribution.

When a publisher wants to connect to a Diffusion server it must specify the host and port of a client connector at the server it wants to connect to and then subscribe to topics on that connection (that is, to behave exactly like any other client).

A publisher can be caused to make such server connections either by configuring them in `etc/Publishers.xml` or programmatically using `Publisher.addServerConnection`.

Android clients

This is an Android client which adheres to the Diffusion API. It has been written in Java using the Android SDK specification.

The client is configured to communicate with a Diffusion server using DPT, or over a secure DPTS connection.

iOS clients

This is a mobile client which adheres to the Diffusion API and has been written as a Objective C library.

The client is configured to communicate with a Diffusion server using DPT or WebSockets, or over a secure DPTS or WSS connection.

Client close reason

There are many reasons why the client can be closed. These are expressed by the ClientCloseReason enumeration and described here.

CONNECTION_LOST

The connection to the client was lost. The connection might have been dropped by the client. Recoverable. A client can be closed for many reasons that are presented as CONNECTION_LOST. During connection the connection can be lost. Some possible reasons are included in the following list:

- The server might have received a connection or reconnection request from a client already connected.
- The server might have received a reconnection request without a client ID.
- The connection might not have been authorized because the credentials are wrong.
- The maximum number of clients might already be connected.

Once connected the connection can be lost for different reasons. Some possible reasons are included in the following list:

- If messages time out while waiting in the connector output buffer. This can happen if a client is slow consuming and the connector output buffer is larger than the TCP output buffer.
- If the client closes its connection while the server is writing a message to the client.
- With the chunked encoding based connection the HTTP response is completed by the server. If the client does not open a new request within a timeout, the clientis closed.
- If a poll request times out and the server finds that the connection has already been closed by the client.

IO_EXCEPTION

An unexpected IO Exception occurred. Recoverable. While trying to perform an I/O operation an exception was generated. This often means that Diffusion attempted to read from a closed TCP connection. This can also happen to HTTP Comet streaming clients if there is a malformed chunk length header before a chunk of messages. This might happen if a message that is larger than the input buffer of the connector is received. The message cannot be read by Diffusion so the client is closed. When Diffusion is handling SSL connections if there is a problem encrypting or decrypting a message, the client is closed for this reason.

MESSAGE_QUEUE_LIMIT_REACHED

The maximum outbound queue size was reached for the client. Not recoverable. Messages sent to the client are placed in a queue. This queue has a maximum allowed size. If the queue limit is reached the client is closed and the queue discarded. The queue is intended to protect against slow patches, reaching the queue limit is taken to mean that the client cannot keep up with the number of messages sent to it.

CLOSED_BY_CLIENT

The client requested close. Not recoverable (unless TEST_RECONNECT is true).

MESSAGE_TOO_LARGE

The client sent a message that exceeded the maximum message size. The server has a maximum message size. If a client sends a message larger than this the server is unable to process it. When this happens the message is discarded and the client is closed.

INTERNAL_ERROR

An internal error occurred.

INVALID_INBOUND_MESSAGE

An inbound message with an invalid format was received. A message received by the server is not a valid Diffusion message. The server is unable to process this and closes the client that sent it.

ABORTED

The client connection was aborted by the server. This is might be because the connection was disallowed. Abort messages are also sent to clients that have unrecognized client IDs. This might be because the server closed the client previously but the client is unaware of this and tried to continue interacting with the server.

LOST_MESSAGES

Loss of messages from the client has been detected. For example, whilst waiting for the arrival of missing messages in a sequence of messages a timeout has occurred.

HTTP based transports use multiple TCP connections. This can cause the messages to be received out of order. To reorder the messages those sent to the server can contain a sequence number indicating the correct order.

If a message is received out of order there is a short time for the earlier messages to be received. If the messages are not received in this time the client is closed.

Missing or invalid sequence numbers also close the client for this reason.

Messages containing a sequence number which has previously been seen are ignored.

This cannot be recovered from as the client and the server are in inconsistent states.

SERVER_CLOSING

The client was closed as part of the Diffusion shutdown process.

CLOSED_BY_PUBLISHER

A publisher initiated the client close.

CLIENT_UNRESPONSIVE

The client has either failed to respond to a ping message in a timely manner or the client has failed to open an HTTP poll for messages. The client does not appear to receive messages.

Cross domain

Cross domain grants permission to communicate with servers other than the one the client is hosted on.

Cross-domain XML file

The cross-domain policy is defined in an XML file

A cross-domain policy file is an XML document that grants a web client – such as Adobe Flash Player, Adobe Reader, Silverlight Player – permission to handle data across multiple domains. When a client hosts content from a particular source domain and that content makes requests directed towards a domain other than its own, the remote domain must host a cross-domain policy file that grants access to the source domain, allowing the client to continue with the transaction. Policy files grant read access to data, permit a client to include custom headers in cross-domain requests, and are also used with sockets to grant permissions for socket-based connections.

For example, say that the Diffusion client is loaded from static.example.com and the connection URL to the Diffusion client is `http://streaming.example.com`, a `crossdomain.xml` file must be loaded from `static.example.com`

A `crossdomain.xml` is required if one of the following is true:

- You are using Diffusion as a streaming data server and a separate web server which are on different domains
- The Diffusion connection type is *HTTP*, *HTTPS*, *HTTPC*, or *HTTPCS*
- You are not using a load balancer to HTTP rewrite Diffusion traffic

Installing the `crossdomain.xml` file for Flash/Silverlight HTTP request

- If you use Diffusion as a web server, copy the `crossdomain.xml` file from the Diffusion install `/etc` folder to the root of the `html` folder

- If you do not use Diffusion as a web server, copy the `crossdomain.xml` file from the Diffusion install `/etc` folder to the virtual root of the web server hosting the Diffusion html lib folder

By default, Diffusion does not have `crossdomain.xml` installed. We shipped an example which allow all domains and all ports to access the Diffusion server. This must be edited to include the correct security details for your installation.

Flash security model

How Flash interacts with remote services to establish security

If a socket-based connection is to be used, for example Diffusion DPT type connection, the Flash player tries to get a policy file from the same host as you are trying to connect to but on port 843. If this port is not open through your firewalls or is not configured within the Diffusion connectors, the Flash player waits 2 seconds before requesting a policy file from the same port that you are trying to connect to. If the policy file request is not responded to correctly or the policy file has restricted the connection, the Flash player generates a security exception and the connection attempt stops.

If an HTTP connection is to be used, for example Diffusion HTTP type connection, a socket-based policy file is not required but a `crossdomain.xml` file might be required before the Diffusion connection is made.

Official Adobe documentation is available at the following location: Cross-domain policy file specification.

FlashPolicy.xml file

When is the FlashPolicy.xml used?

When a Diffusion *DPT* connection is used a socket connection is made, in order that the socket connection can be established a socket policy file must be acquired from port 843 or from the port that the Diffusion client is trying to connect to.

Again this is part of the cross-domain schema, but this time the `to-ports` attribute on the `allow-access-from` element is particularly important.

FlashMasterPolicy.xml file

Use of the FlashMasterPolicy file

FlashMasterPolicy is used for requests on port 843. It is a normal `crossdomain.xml` with an extra element of

```
<site-control permitted-cross-domain-policies="master-only" />
```

The `site-control` element here specifies that only this master policy file is considered valid on this domain

Silverlight security model

How Silverlight interacts with remote services to establish security

If a socket-based connection is to be used, for example Diffusion DPT type connection, the Silverlight player tries to get a policy file from the 943. If this port is not open through your firewalls or is not configured within the Diffusion connectors the Silverlight player generates a security exception and the connection attempt ceases.

If an HTTP connection is to be used, for example. Diffusion HTTP type connection, a socket-based policy file is not required but a `crossdomain.xml` file might be required before the Diffusion connection is made.

Official Microsoft documentation:

Network security access restrictions (Silverlight)

Silverlight `clientaccesspolicy.xml` file

When is the `clientaccesspolicy.xml` used?

When a Diffusion *DPT* connection is used a socket connection is made, in order that the socket connection can be established a socket policy file must be acquired from port 943.

JavaScript security model

How JavaScript interacts with remote services to establish security

The Diffusion JavaScript client library, unless otherwise configured, cascades downward through a set of transports starting with WebSockets and working its way down toward XmlHttpRequest (also known as XHR or Ajax Long Poll), and finally to hidden frames.

WebSockets, Flash, and Silverlight have few security constraints, however XHR is subject to the same origin policy. Simply put, if JavaScript code executes within a web page sourced from `www.example.com` it is only permitted to make XHR requests back to `www.example.com`. If your Diffusion server is at `push.example.com` this presents a problem when only XHR is available.

The catch-all solution

The set of web browsers in current use is both broad and heterogeneous. Rather than catering to each special case browser, this approach contains all complexity to one place: the load balancer. This presumes there is a load-balancer however, though in all reasonable production circumstances this is true. All XHR requests to Diffusion use a URL that starts /diffusion. Routing all such requests to one of the servers in the Diffusion pool will make available both regular and Diffusion functionality from one apparent host. This approach is suitable to all web browsers.

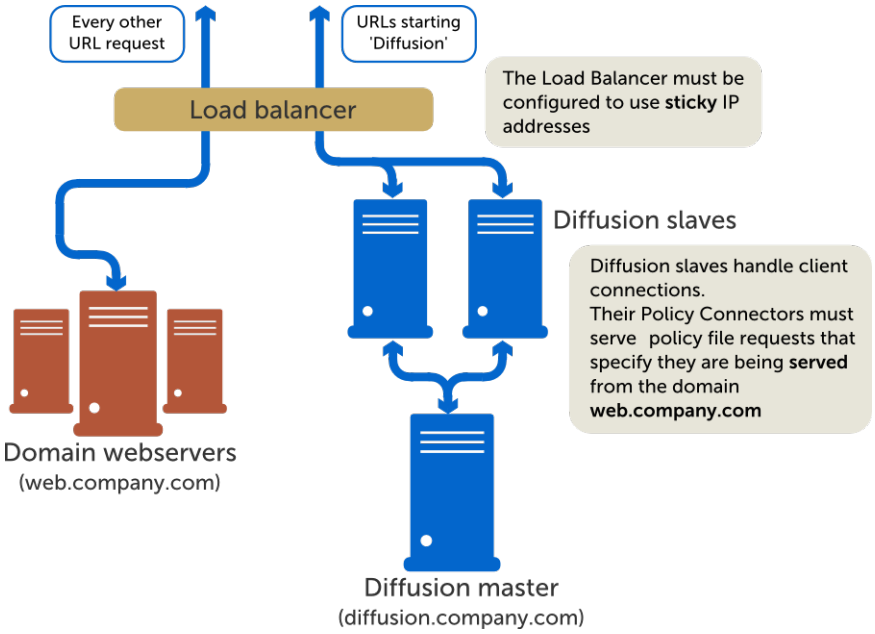

Every other URL request
URLs starting 'Diffusion'
Load balancer
The Load Balancer must be configured to use **sticky** IP addresses
Diffusion slaves
Diffusion slaves handle client connections.
Their Policy Connectors must serve policy file requests that specify they are being **served** from the domain **web.company.com**
Domain webservers
(web.company.com)
Diffusion master
(diffusion.company.com)

Figure 13: Using a load balancer to composite two URL spaces into one.

In circumstances where clients of Diffusion solutions cannot be depended upon to have a single IP address (for example: users with multiple aDSL connections, or smart-phones migrating between providers), each HTTP request made from a Diffusion client to a Diffusion server holds a cookie named "session" holding the unique client-id of that client. This gives load balancers an alternative means of distributing a request to one of their Diffusion server team. Further details can be found in the Configuration section of the manual.

Software alternatives

For test and development purposes a hardware load balancer might be an expensive means of compositing the URL-spaces of two (or more) web servers into one. Alternatives such a mod_proxy for the Apache web server, and an ISAPI_Rewrite tool can be employed to achieve the same for a lower (or zero) price-tag. Diffusion can be configured to run as a servlet with Tomcat. For more information, see Web server installation on page 53

CORS

Cross-Origin Resource Sharing is a standard formed to address circumstances where www.example.com uses XHR to access resources on alternate host push.whatnot.com, and aims to provide sensible constraints and avoid a free-for-all.

Client-Side

Include the XHRURL attribute in the arguments to the DiffusionClientConnectionDetails constructor. For example:

```
var connectionDetails = {
          debug : true,
          onDataFunction : onDataEvent,
          XHRURL: "http://www.example.com:8080"
          topic : 'SYMBOLS/QUOTES/NIFTY~INDEX',
}
```

Server side

CORS filtering is governed on the server side using the cors-origin attribute found in `etc/WebServer.xml`. By default this is a very permissive `.*` regular expression, and must be set to something more specific in production. In the above example, `push.example.com` will limit requests to `push.example.com` to only those from `www.example.com`. Full details about this feature are found in the web server section of the Diffusion manual.

Discussion

CORS support is currently lacking from Internet Explorer®, making it useless in most internet-facing production scenarios.

Protocols supported

Each client supports varying transports. A table of the supported transports for each client is presented here.

The APIs table describes a more detailed list of the implemented protocols. Some protocols have been partially implemented and are not supported. Support for J2ME and C is on a best effort basis. For more information, see Table of APIs on page 343.

The JavaScript client is fully supported only on certain browsers. Best effort support is provided for other browsers but the software/hardware combination might not be reproducible, particularly for mobile browsers. Refer to Browsers supported on page 118 for detail about the supported browsers.

Refer to Protocols overview on page 604 for more information about the protocols.

The following table lists the protocols supported for each client:

Table 19: Supported protocols by client

Client	DPT	DPTS	WS	WSS	HTTP Full Duplex	HTTPS Full Duplex	HTTP Chunked Streaming	HTTPS Chunked Streaming	HTTP Polling	HTTPS Polling
Java	✓	✓	✓	✓	✓	✓				
.NET	✓	✓	✓	✓	✓					
JavaScript	✓ [1]	✓ [1]	✓	✓			✓ [2]	✓ [2]	✓	✓
Flash	✓	✓					✓	✓	✓	✓
Silverlight	✓	✓							✓	✓
iOS	✓	✓	✓	✓						
Android	✓	✓								
C	✓									

Interfaces supported

Diffusion provides APIs in a variety of languages. Each of these APIs is supported over a set of protocols and in a specific level of the language.

Supported APIs in the Classic API

	Publisher	Event publisher	Client	Implementation version	Protocols
Java	YES	YES	YES	Java 7 or Java 8	• DPT, DPTS • WS, WSS • HTTP, HTTPS (Full duplex)
.NET		YES	YES	.NET 3.5	• DPT, DPTS • WS, WSS • HTTP (Full duplex)
JavaScript			YES	JavaScript V1.3	• Native: WS, WSS, HTTP, HTTPS, HTTP Streaming

[1] Supported by Flash/Silverlight
[2] Supported natively and by Flash

	Publisher	Event publisher	Client	Implementation version	Protocols
					• Flash: DPT, HTTP, HTTPS, HTTP Streaming • Silverlight: DPT, HTTP, HTTPS
Flash			YES	ActionScript V3.0 (Flex 3.0)	• DPT, DPTS • HTTP, HTTPS • HTTPC, HTTPCS
Silverlight			YES	Silverlight V4.0	• DPT • HTTP, HTTPS • HTTPC, HTTPCS
iOS			YES	iOS v5.1.1, architectures: armv7, armv7s, arm64	• DPT, DPTS • WS, WSS
Android			YES	Android 2.2	• DPT, DPTS

Tier 2 APIs supported on a best effort basis

	Publisher	Event publisher	Client	Implementatio version	>Protocols
C			YES		• DPT

For more information on the transports supported by the clients please refer to the section Transports supported

Supported APIs in the Unified API

Table 20: Tier 1 supported client platforms

Platform	Supported version	Supported transport protocols
Java	7, 8	• DPT, DPTS • WS, WSS • HTTP, HTTPS (Full duplex)
.NET	3.5	• DPT, DPTS • WS, WSS • HTTP (Full duplex)
C	-	• DPT

Browsers supported

Some of the client libraries are intended to be run within browser environments. Diffusion can use most commercial browsers and their variants. However, some Diffusion API protocols might not be available.

Diffusion supports the latest release of the following browsers based on the original Diffusion release date.

Table 21: Supported browsers

Browser	Version
Google Chrome™	36.0
Mozilla Firefox®	31
Microsoft Internet Explorer	11.0
Apple Safari®	7.0.6

- Push Technology runs full regression tests on the browsers and versions documented above for every patch release.
- Push Technology also carries out basic validation testing on the latest versions of these browsers but full support is available only at the next minor release. Support for untested browsers is provided on a best effort basis.
- Support for older versions of browsers is also provided on a best-effort basis, unless specified otherwise.
- Support for other browsers is also provided on a best-effort basis.

Browser plugins (Flash and Silverlight)

Diffusion only supports official Flash and Silverlight players.

Table 22: Browser Plugins

Plugin player	Version
Adobe Flash Player	11.6
Microsoft Silverlight	5.1

For details about the operating systems and browsers supported by the Silverlight plugin, refer to the "System Requirements" section on the following web page: Get Microsoft Silverlight

Browser limitations

Some browsers cannot use all the Diffusion protocols or features. If you experience problems when developing with protocols or client libraries that use the browser, check whether the browser supports this function.

Browser environments are not uniform and some browsers might have functional limitations. It is important to be aware of these limitations when developing for compliance with target browsers.

WebSocket limitations

WebSocket is an Internet Engineering Task Force (IETF) protocol used by Diffusion to provide a full-duplex communication channel over a single TCP connection. It is not supported by all browser versions.

- Internet Explorer
- Firefox
- Chrome
- Safari
- Opera®
- iOS
- Android

Table 23: Internet Explorer support for WebSocket

Version	WebSocket support?
7.0	NO
8.0	NO
9.0	NO
10.0	YES
11.0	YES

Table 24: Firefox support for WebSocket

Version	WebSocket support?
27.0	YES
28.0	YES
29.0	YES
30.0	YES
31.0	YES

Table 25: Chrome support for WebSocket

Version	WebSocket support?
32.0	YES
33.0	YES
34.0	YES
35.0	YES
36.0	YES

Table 26: Safari support for WebSocket

Version	WebSocket support?
5.1	YES
6.0	YES
6.1	YES

Version	WebSocket support?
7.0	YES
8.0	YES

Table 27: Opera support for WebSocket

Version	WebSocket support?
19.0	YES
20.0	YES
21.0	YES
22.0	YES
23.0	YES

Table 28: iOS support for WebSocket

Version	WebSocket support?
5.0	NO
6.0	YES
6.1	YES
7.0	YES
7.1	YES

Table 29: Android support for WebSocket

Version	WebSocket support?
4.0	NO
4.1	NO
4.2	NO
4.3	NO
4.4	YES

Cross-origin resource sharing limitations

Cross-origin resource sharing (CORS) allows resources to be accessed by a web page from a different domain. Some browsers do not support this capability.

Cross-origin resource sharing uses HTTP headers to enable the Diffusion server to indicate if it accepts traffic from web pages served from other servers. When a CORS request is made, Diffusion must respond with certain response HTTP headers for the browser to treat the request as successful. CORS requests can result in the browser sending a pre-flight request to Diffusion using the OPTIONS method to determine if the origin, headers, and methods of the request it is about to make are permitted. Diffusion responds with the correct values for headers and methods but the actual request is not made until the pre-flight request succeeds. The allowed origins can be configured in the client-service element of the `WebServer.xml` configuration file.

- Internet Explorer
- Firefox
- Chrome
- Safari

- Opera
- iOS
- Android

Table 30: Internet Explorer support for CORS

Version	CORS support?
7.0	NO
8.0	NO
9.0	NO
10.0	YES
11.0	YES

Table 31: Firefox support for CORS

Version	CORS support?
27.0	YES
28.0	YES
29.0	YES
30.0	YES
31.0	YES

Table 32: Chrome support for CORS

Version	CORS support?
32.0	YES
33.0	YES
34.0	YES
35.0	YES
36.0	YES

Table 33: Safari support for CORS

Version	CORS support?
5.1	YES
6.0	YES
6.1	YES
7.0	YES
8.0	YES

Table 34: Opera support for CORS

Version	CORS support?
19.0	YES
20.0	YES

Version	CORS support?
21.0	YES
22.0	YES
23.0	YES

Table 35: iOS support for CORS

Version	CORS support?
5.0	YES
6.0	YES
6.1	YES
7.0	YES
7.1	YES

Table 36: Android support for CORS

Version	CORS support?
4.0	YES
4.1	YES
4.2	YES
4.3	YES
4.4	YES

Browser connection limitations

Browsers limit the number of HTTP connections with the same domain name. This restriction is defined in the HTTP specification (RFC2616). Most modern browsers allow six connections per domain. Most older browsers allow only two connections per domain.

The HTTP 1.1 protocol states that single-user clients should not maintain more than two connections with any server or proxy. This is the reason for browser limits. For more information, see RFC 2616 – Hypertext Transfer Protocol, section 8 – Connections.

Modern browsers are less restrictive than this, allowing a larger number of connections. The RFC does not specify how to prevent the limit being exceeded. Either connections can be blocked from opening or existing connections can be closed.

- Internet Explorer
- Firefox
- Chrome
- Safari
- Opera
- iOS
- Android

Table 37: Internet Explorer maximum supported connections

Version	Maximum connections
7.0	2

Version	Maximum connections
8.0	6
9.0	6
10.0	8
11.0	13

Table 38: Firefox maximum supported connections

Version	Maximum connections
27.0	6
28.0	6
29.0	6
30.0	6
31.0	6

Table 39: Chrome maximum supported connections

Version	Maximum connections
32.0	6
33.0	6
34.0	6
35.0	6
36.0	6

Table 40: Safari maximum supported connections

Version	Maximum connections
5.1	6
6.0	6
6.1	6
7.0	6
8.0	6

Table 41: Opera maximum supported connections

Version	Maximum connections
19.0	6
20.0	6
21.0	6

Version	Maximum connections
22.0	6
23.0	6

Table 42: iOS maximum supported connections

Version	Maximum connections
4.0	4
5.0	6
6.0	6
6.1	6
7.0	6
7.1	6

Table 43: Android maximum supported connections

Version	Maximum connections
4.0	6
4.1	6
4.2	6
4.3	6
4.4	6

Diffusion protocols like HTTP, HTTPS, HTTPC and HTTPCS use up to two simultaneous connections per Diffusion client. It is important to understand that the maximum number of connections is per browser and not per browser tab. Attempting to run multiple clients within the same browser might cause this limit to be reached.

Reconnection can be used to maintain a larger number of clients than is usually allowed. When TCP connections for HTTP requests are closed, the Diffusion sends another HTTP request which the server accepts. Be aware of cases where Diffusion tries to write a response to closed polling connections before the client can re-establish them. This behavior results in an IO Exception and the Diffusion server closes the client unless reconnection is enabled. When the client tries to re-establish the poll, it is aborted.

Another way to get around browser limits is by providing multiple subdomains. Each subdomain is allowed the maximum number of connections. By using numbered subdomains, a client can pick a random subdomain to connect to. Where the DNS server allows subdomains matching a pattern to be resolved as the same server, tab limits can be mitigated substantially.

Plugin limitations

When plugins such as Flash or Silverlight make HTTP requests they go through the browser instead of making them directly. This means they do not avoid browser limits on the number of connections and must wait for the browser to make the request. There is an additional layer between the client and the connection.

Browser buffering

When streaming, such as with HTTPC connections, the web browser might not release the data it receives immediately. Browsers might require that a minimum amount of data is received before passing the data on to the client. After this minimum amount has been received future messages are passed by the browser as soon as the browser receives them. This might cause problems when there is a delay between the initial topic load and delta messages. To handle this the first message sent over a HTTPC poll is padded with null bytes to fill this buffer. The amount of padding sent can be configured with the comet-initial-message-padding configuration item (see WebServer). The amount of padding required varies between browsers.

Chapter 6

Topics

In this section:

- Topic basics
- Topic tree
- Topic naming
- Topic aliasing
- Creating topics
- Topic subscription
- Topic selection
- Topic loading
- Topic data
- Topic fetch
- Topic sets
- Topic attachments

Diffusion publishes messages to topics, to which clients can subscribe. All clients subscribed to a topic receive all messages published to the topic. The topic is a fundamental part of Diffusion.

Topic basics

This section introduces the concept of topics within Diffusion.

What is a topic?

Diffusion is essentially a publish/subscribe message broker.

Publish/subscribe (or pub/sub) is an asynchronous messaging paradigm where senders (publishers) of messages do not send their messages to specific receivers (subscribers). Rather, published messages are characterized into topics, without knowledge of what (if any) subscribers there might be. Subscribers express interest in one or more topics, and receive only messages that are of interest without knowledge of what (if any) publishers there are. This decoupling of publishers from subscribers allows for much greater scalability and a more dynamic network topology.

In Diffusion a topic has state, unlike the queues/topics of a message-broker. It is a named object that is registered (or added) in a Diffusion Server by a Publisher and enables not only basic pub/sub messaging but also a number of other features which are introduced below.

How are topics used?

Topics can be used in a number of ways.

The main ways in which topics are used within Diffusion are as follows:

Publishing to all subscribed clients

The publisher publishes messages to the topic. Clients subscribe to topics and each client that is subscribed to a topic will receive all messages published to that topic.

Publishing to all but one subscribed client

A publisher can publish a message to a topic, but exclude a specified client.

Sending messages to individual clients

A publisher can send a message to an individual client (or group of clients) which the client receives like any other message delivered for the topic. A client can send a message to a topic which in turn is routed to the publisher.

Client to publisher communication

A client can send a message to a topic which in turn is routed to the publisher.

Client fetch

A client can retrieve the current state of an existing topic without subscribing to it using a fetch request. When a client attempts to fetch the state of a topic that does not exist, a clientFetchInvalid notification occurs which allows the publisher to send a response to a fetch request (using Client.sendFetchReply) for a topic that does not even exist if that is what is required.

Receiving event data

An event publisher can connect to a Diffusion server and send messages to a topic. Such messages are routed to the publisher providing the topic.

Distributing publishers

Publishers can be distributed by means of a publisher connecting to another Diffusion server and subscribing to topics as if it were a client. The communication available between publishers is provided through topics in exactly the same way as communication between publishers and clients in general.

Topic trees

Topics are arranged in a tree

In the simplest case topics are flat in structure. For example, a single publisher provides three topics called “A”, “B” and “C” which are subscribed to individually.

Topics can also be organized in a topic tree, so that topics have subordinate topics and form a tree of topics. This allows for much more control over the topics. For example, you can subscribe to branches of the tree rather than individual topics like “foo/bar//”. To better manage this control, subordinate topics cannot be created by publishers that did not create the parent topic. This prevents subscriptions to branches of the tree from being handled by multiple different publishers.

Subscription

Subscription refers to a client's association with a topic

Subscription is the term used to describe a client registering an interest in a topic such that it receives all messages published to that topic.

A client can subscribe to any number of topics.

When a client subscribes to a topic, typically the first message it receives on that topic is a topic load (see below) which provides the current data state of that topic. It then receives all updates to the topic as delta messages.

A client can subscribe to a topic using an explicit topic name or can subscribe to many topics using a topic selector.

A client can subscribe to a topic that does not exist. The client is not be informed that the topic does not exist. However, publishers can be notified that a client has attempted to do this. When a publisher creates a new topic, it has the option to subscribe any clients to it that might have tried to subscribe to it previously. When clients subscribe to topics before they are registered and are subscribed later when the topic is registered, it is called pre-emptive, or future subscription.

When a client requests subscription to a topic, that subscription is validated in the following order:

- Authorization handler
- Subscription policy (Blacklist/Whitelist)
- Topic subscription handler

Clients can resubscribe to topics that they are already subscribed to without being doubly subscribed or causing an error. Resubscribing to a topic is a no-op.

Clients can unsubscribe from topics at any time.

Topic loading

The initial state of a topic is referred to as the topic load

When a client subscribes to a topic the publisher is notified. At this point it is usual for the publisher to provide the current data state of the topic in the form of a topic load message. This act of providing the initial topic load data is known as topic loading.

Topic data

Topic data is the data state associated with a topic

In the broadest sense, topic data is best thought of as the data associated with a topic and maintained by the topic provider. This data has an initial state and may then be updated causing delta messages to be sent to clients to notify the changes to the data. New clients subscribing to a topic are typically sent the current state of the topic's data as a topic load (see above).

To simplify the management of topic data the Java API provides built in mechanisms for maintaining and updating topic data and reading and generating various types of message content. For more information see the Topic data section.

Topic attachments

A topic may have any type of object attached to it for processing convenience

The Java API allows an object to be attached to a topic. This might be used to associate some data handling object with the topic when the topic data feature is not appropriate.

Topic references

A alternative name or piece of text may be associated with each topic

Often topic names may not be very meaningful. The Java API allows a reference to be set for a topic which might be used in situations where the topic name might be displayed.

Topic streams

In the Diffusion, a topic stream receives topic events for the topic or topics it is registered against. A topic stream can be registered using a topic selector that matches zero, one, or many topics.

The topic events that the topic stream can receive are the following:

- the client subscribes to the topic
- the client unsubscribes from the topic
- the topic is updated

Topic streams continue to receive notification of topic events until the stream is closed or discarded. A stream is closed when it is no longer required. A stream can be discarded if a problem occurs, for example, if the client session dies. In both the close case and the discard case, the topic stream has the opportunity to do any required clean up of resources or take any required actions.

Topic streams can be invoked in any order and cannot consume topic content or prevent content from being passed to other topic streams or topic listeners.

Topic listeners

An object can be nominated to receive topic events and process inputs for topics.

> **Note:** Topic listeners are used in the Classic API. Topic listeners are also present in the Unified API, but are deprecated in favor of topic streams.

As topic messages arrive at an application (for example, a publisher) they are routed to methods to handle them. However, in addition to the normal recipient method, you can specify additional topic listeners which handles messages for specified topics. Such a listener can handle the messages for only one topic or for many different topics (specified using topic names regular expression patterns).

For example, a publisher can add different listeners to handle message from clients, Event publishers or servers or a client can add listeners to handle different messages from servers.

Topic listeners allow for the processing for particular topics to be easily encapsulated.

Topic listeners are invoked in the order that they are declared and before any fixed listener interface. Any topic listener can choose to consume the message which means that it is not then passed to any other listeners. However, topic streams are separate from topic listeners and if a topic listener consumes topic content, this does not prevent the content being passed to any topic streams.

Topic providers

A topic provider is used to create and manage topics.

A topic provider is able to create and destroy topics. It is not necessary to be a topic provider to send messages to a topic but a topic provider can be used to create a `TopicMessage` object. A topic provider owns the topics that it creates. A topic provider cannot create a subordinate topic to a topic that it does not own. An `TopicInvalidException` is thrown if you attempt to add a topic that is owned by another topic provider. Topic providers can remove topics that they do not own. Every topic knows its topic provider and can get it with `getTopicProvider`.

Topic tree

Topics are organized in a tree structure. Paths to topics consist of level names separated by the slash character (/). A topic can be bound to any level of the hierarchy.

In a very simple case there might be only one publisher deployed in a Diffusion server which has a small number of discrete topics. At the other extreme there might be many publishers, each with many (perhaps thousands of) topics. The number of topics is largely to do with how it is decided to model the data that is published for an application.

When there are very few topics it is easy enough to organize them in a flat structure as follows:

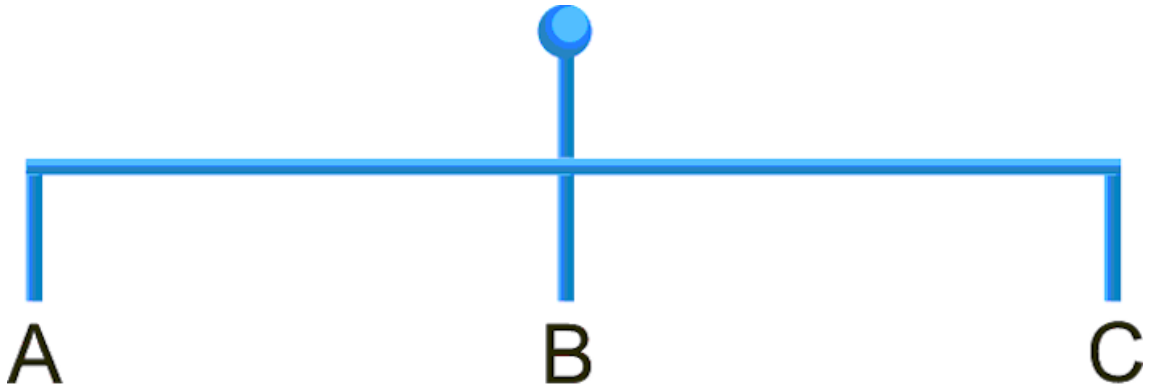


Figure 14: Flat structure

However, when there are a large number of topics it becomes more difficult to handle them as the publisher has to register and load each one and the client has to subscribe to each one. Also it might be difficult to map complex data structures onto topics. Diffusion allows topics to be organized hierarchically to simplify the handling of those topics and also the mapping of data to topics.

An example of a hierarchical topic structure is shown in the following figure.

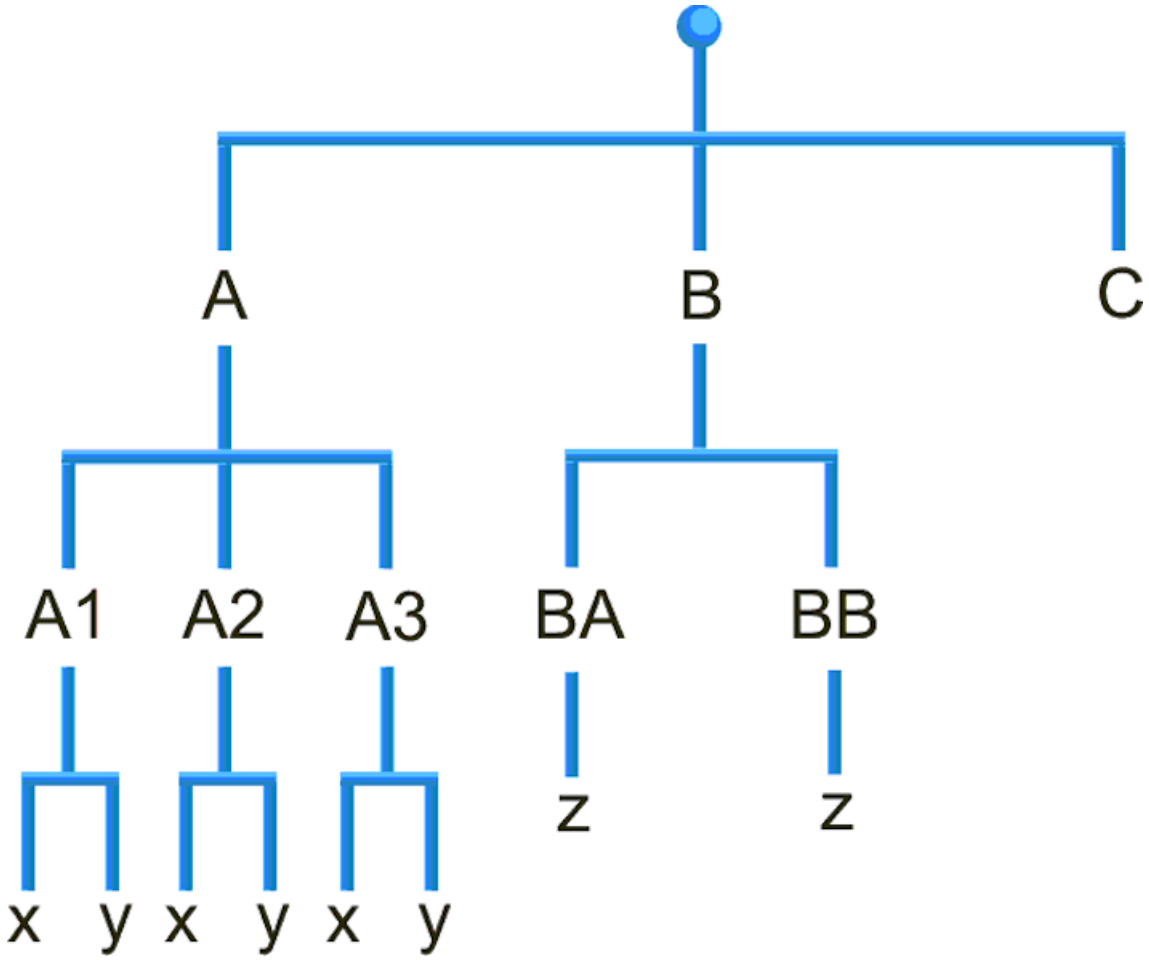


Figure 15: Hierarchical topic structure

In the above case the topics A, B and C can be provided by different publishers or all by the same publisher. The difference from the first examples is that A and B have subordinate topics allowing data to be organized in a more structured way. The topics A, B and C are owned by the publisher providing them and the subordinate topics can only be created by the owning publisher. A publisher cannot create the topic A1 if does not own A.

The topic tree

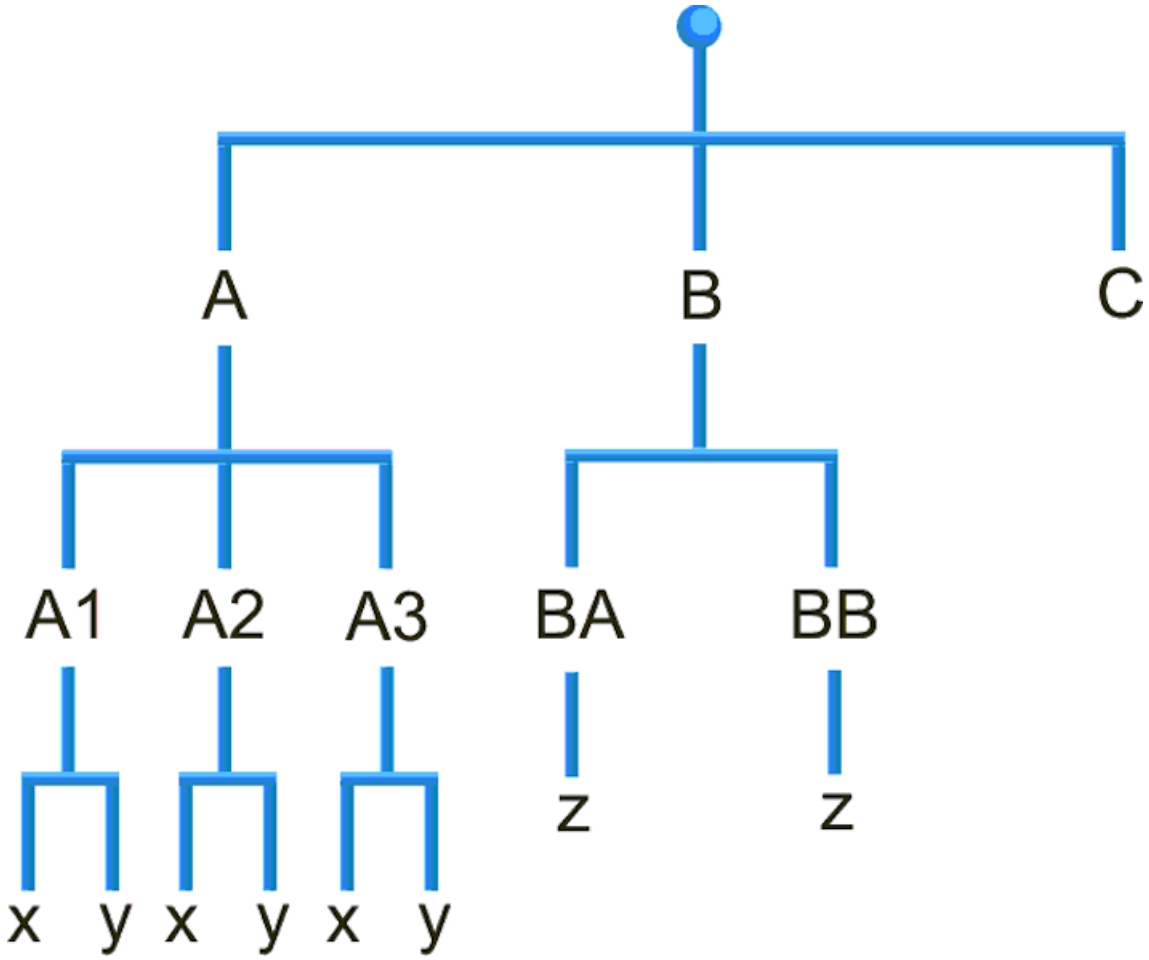
The hierarchy of topics that is present on any one Diffusion server is known as the topic tree. Inside the topic tree are top-level topics with subordinate topics (these subordinate topics can themselves have subordinate topics).

The topic tree can be obtained using the `Publishers.getTopicTree` method which returns a topicTree. Every subordinate node is a topic. All nodes are of type TopicTreeNode. These classes provide comprehensive mechanisms for navigating the tree structure. The TopicTree node has a fast topic lookup facility in its getTopic method.

Each TopicTreeNode is owned by the topic provider that created it. The ownership of a topic is distinct from the parent of a topic. They are related by controlling which publisher can create a child topic node on the topic tree.

Referencing hierarchic topics

A hierarchic topic can be accessed using its path or full hierarchic name which is represented as the name of each topic node within it, hierarchy separated by the slash character (/). It is this name that is used at subscription and it is this name that is carried within Messages.



For example, in the above tree the names of the topics in the branch headed by the topic called 'A' are:

- `A`
- `A/A1`
- `A/A1/X`
- `A/A1/Y`
- `A/A2`
- `A/A2/X`
- `A/A2/Y`
- `A/A3`
- `A/A3/X`
- `A/A3/Y`

Referencing segments of the topic tree

Topics can be referenced individually by their full path name but in many situations there is also the facility to refer to branches of the topic tree or fuzzy match with topic names. This is fully described in topic selection.

Subscribe all subordinates

For example, to subscribe to topic “A” above and all its subordinates using the Java client API:

```
ExternalClientConnection connection =
 new ExternalClientConnection(this,"dpt://MyHost:8080");
connection.connect();
```

```
connection.subscribe("A//");
```

Subscribe only children

Or to subscribe to only the topics (X and Y) subordinate to the A2 topic

```
connection.subscribe("A2/");
```

Topic naming

This section covers all aspects of topic naming including recommendations of how to name your topics.

A topic name can be made up of any number of Unicode characters but must not contain any of the restricted characters mentioned below. When a topic tree is used, the topic path is made up of the names of all topics in its path separated by the slash character (/).

Restricted characters

The following characters are not permitted in topic names:

Table 44: Restricted characters

Character	Reason for restriction
/	This is a name separator and cannot occur in a topic name.
[]\^$.|?*+()	These are all metacharacters used in regular expressions. Any topic String that contains any of these characters is assumed to be a topic selector. These characters cannot be used in topic names.
Control/Reserved	No characters with a hexadecimal value of less than 0x2D. This includes some punctuation characters such as comma (,).
Whitespace	No characters defined as whitespace in Java (as indicated by the isWhiteSpace method of the Java Character class).

Recommendations

Although all unicode characters (other than the restricted ones mentioned above) are supported to allow for language variations it is highly recommended that only alphanumeric characters are ever used in topic names with. Hyphen (-) or underscore (_) can be used as break characters.

Generally speaking, keep topic names short as every topic message has to carry the full topic name which can potentially have significant message size implications. However, the optional Topic aliasing feature can significantly reduce the size of the topic names transmitted in messages which can be very important, especially when hierarchic topics are in use.

Topic references

Within a publisher a topic can be assigned a more meaningful reference than its topic name. This can be helpful in identifying the topic in logs etc, especially when short unique topic names are used.

A topic reference can be set using the setReference method and obtained using getReference.

Topic aliasing

You can save space in messages by using topic aliases to shorten long topic names.

It is important to remember that the topic name forms part of each message header, so the smaller the topic name the better. However, when hierarchic topics are in use it can be difficult to keep topic names short.

Topic aliases are a mechanism for automatically using short topic names in all delta messages. It is an optional feature, but turned on by default.

The use of topic aliases can save a significant amount of network traffic, however there is a trade off of a small processing cost which can reduce the message throughput.

Topic aliasing can be turned on or off for a publisher using a `etc/Publishers.xml` entry. For an event publisher, it can be turned on or off using a `setTopicAliasing` method.

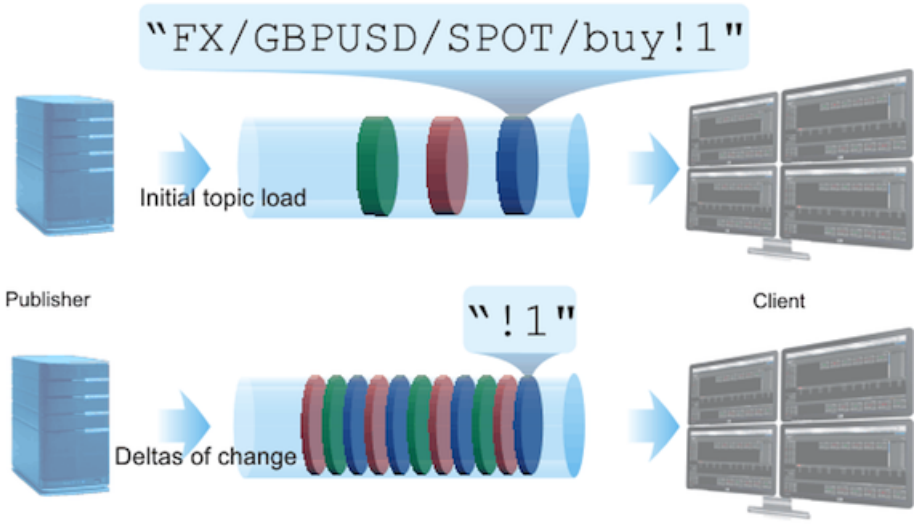


Figure 16: Topic aliasing

When turned on, the creation of topic load messages causes short topic aliases to be generated for all topic names greater than 4 characters in length. Delta messages for topics with aliases contain only the topic alias, which saves space in each message sent.

> **Warning:** If aliasing is turned on, creating the load message without sending it to clients still results in delta messages being aliased – this can lead to failures at the client.

You can override the publisher aliasing requirement on a per topic basis by creating the topic with a TopicDefinition and explicitly specifying the aliasing property.

The use of topic Aliases is totally transparent to the API where topic names are always presented in full. However, the diagnostic display of messages might show an alias (a short ID prefix by an exclamation mark (!) character).

> **Note:** If topic load messages are not used for a topic, an alias is not generated for that topic and the saving is achieved only for topics that send a topic load message to the client on subscription. Likewise, unless the first message sent by an event publisher for any topic is a topic load, subsequent deltas do not use aliases.

Creating topics

Topics are created by publishers to make them available for clients for subscription.

Topics can be created on behalf of a publisher in the xml configuration files or by creating them dynamically from within the publisher.

Configuring topics

How to specify topics in `publishers.xml`

When a publisher provides a relatively small number of topics (or you want it to start with a small number of topics), one way of creating topics is by configuring them for the publisher in `etc/Publishers.xml`

One or more topics can be named in the configuration and these topics are automatically added during publisher startup.

Any topics specified in this way are also re-added if the publisher were stopped and restarted.

Topics added in this way are available for subscription to immediately and therefore the data required for the topic must be set up during subscription.

This method of specifying topics using configuration is not compatible with the use of topic data and therefore has limited use.

Dynamically adding topics

How to create topics from within a publisher

The most flexible way of creating topics is dynamically within the publisher. This way a publisher can start up without a full set of topics but can add them as and when required.

This method can still be used along with configured initial topics. For example, you might want the publisher to start with some fixed control topic and then others are added according to what is required.

The following set of methods are available to a publisher for adding topics:

Table 45: Methods for adding topics to publishers

Method	Description
`addTopic(String)` `addTopic(String,TopicTreeNode)`	This is the simplest method where a top level topic of a given name is registered. This can also be used to add a hierarchic topic by specifying a hierarchic name and any missing intermediate topics are also registered. It is however more normal to add hierarchic topics using the second method where the parent topic is specified.
`addTopic(String,TopicData)` `addTopic(String,TopicTreeNode,TopicData)`	As "1" but allowing TopicData to be associated with the topic. The TopicData must have been pre-created using `TopicDataFactory` and configured as required.
`addTopic(String,TopicDefinition)` `addTopic(String,TopicTreeNode,TopicDefinition)`	As "1" but allowing all details of the topic and its TopicData to be specified.

Method	Description
	This is the most powerful method of creating topics and can be used to create topics using the definition from another topic.
`addTopics(TopicSet)` `addTopics(TopicSet,TopicTreeNode)`	Allows a set of topics to be registered. Topic data or attachments cannot be specified when multiple topics are registered at once.

Topics themselves also have similar methods so that child topics can be easily added without having to specify the full hierarchic topic names as in the above examples.

Removing topics

The effects of removing topics

When a publisher is stopped all of its topics are automatically removed, however individual topics can be removed at any time by a publisher.

When a topic is removed all clients that are subscribed to it are automatically unsubscribed.

When a topic is removed, all topics that are subordinate to it in the topic hierarchy are also removed.

Topic subscription

Subscription refers to a client registering an interest in a topic such that it will automatically receive all messages published to that topic.

A client can subscribe to any number of topics and a topic can be subscribed to by any number of clients.

Subscribing to a topic also allows a client to send messages to the publisher of that topic.

Topic subscriptions totally decouple clients and publishers. A client has no knowledge of the publisher or publishers of the topics it subscribes to.

Subscription on connection

A client can subscribe to any number of topics when it connects to the Diffusion server, or it can connect with no topics at all and subscribe later.

The way in which topics are specified on connection varies according to the client API in use. The following example shows subscription on connection in the Java client API:

```
ExternalClientConnection connection =
 new ExternalClientConnection(this,"dpt://localhost:8080");
connection.connect("TopicA","TopicB");
```

Note: The connect method is a varargs method so any number of topics can be specified. There is also a variant that takes a TopicSet.

Ad-hoc subscription

A client can subscribe to any number of additional topics after it has connected.

The following example shows subscription after connection in the Java client API:

```
ExternalClientConnection connection =
```

```
 new ExternalClientConnection(this,"dpt://localhost:8080");
connection.connect();
//...
//...
connection.subscribe("TopicA","TopicB");
```

> **Note:** The subscribe method is a varargs method so any number of topics can be specified. There is also a variant that takes a TopicSet.

Effect of subscription

When a client subscribes to a topic, the publisher of that topic is notified through its subscription method. Typically the publisher sends a topic load message containing the current state of the topic's data to the client.

From that point on all (delta) messages published by the publisher to the topic are consequently sent to the client, until the client either disconnects or unsubscribes from the topic.

Clients of different types receive messages in different ways. In the Java client API a message is received through the `messageFromServer` method on the `ServerConnectionListener` interface. Different listeners can be assigned to different topics as required.

Subscribing to topics that do not exist

If a client attempts to subscribe to a topic that does not exist, any publisher that is listening for client notifications is notified via the `clientSubscriptionInvalid` method. This enables a publisher to provide topics on demand as (assuming it is appropriate to provide this topic) it can add the topic and subscribe the client to it using the `Client.subscribe` method.

When a client subscribes to a topic that does not exist the client is not notified in any way and is not aware that this has happened.

Subscription using topic selectors

A client can subscribe to specific topics but can also use Topic Selection when subscribing to subscribe to many topics at once. Depending upon the selector format the client can easily subscribe to a branch of the topic tree or to a filtered view of the topics.

> **Note:** When selectors are used publishers are not notified of attempts to subscribe to topics that do not exist.

Subscribe all subordinates

So, in the Java client API the client can subscribe to a topic called “A” and all of its subordinate topics as follows:

```
connection.subscribe("A//");
```

Subscribe to top-level topic

Subscribe to all topics named “B” that are subordinate to any top level topic that starts with “A” (for example, “Accounts/B” or “Admin/B”) as follows:

```
connection.subscribe("A+/B");
```

Subscribe multiple selectors

Multiple selectors can be supplied and mixed with real topic names, for example

```
connection.subscribe("A//","MyTopic","A+/B");
```

Subscription using topic sets

At times it might be more convenient to use topic sets to specify the topics to be subscribed to.

The following example shows how this can be done in the Java client API:

```
TopicSet topicSet = new TopicSet("A//","MyTopic");
//.
topicSet.add("A+/B");
connection.subscribe(topicSet);
```

Forced subscription

A publisher can force a client to become subscribed to a topic even if the client has not requested subscription to the topic.

Forced subscribe individual client

To force subscribe an individual client to a topic a publisher can use the `Client.subscribe` method with the force flag set to true. The following example shows a publisher that is listening for client notifications force subscribing all clients that connect to its topic:

```
public void clientConnected(Client client) throws APIException {
 Topic topic = getTopic("MyTopic");
 client.subscribe(topic,true);
}
```

Forced subscribe by topic set

The subscription can equally be done using the topic name, or even multiple topic names by means of a TopicSet as follows:

```
client.subscribe(new TopicSet("MyTopic","AnotherTopic"),true);
```

Forced subscribe all clients

You can subscribe all connected clients to a topic from within a publisher as follows:

```
Topic topic = getTopic("MyTopic");
subscribeClients(topic,true);
```

Resubscription

If a client subscribes to a topic that it is already subscribed to, it does not cause a an error or cause the client to be doubly subscribed. Resubscription is a no-op.

Unsubscribing

A client can unsubscribe from a topic to which it is subscribed to at any time and from that point on no longer receives messages for that topic.

The method for unsubscribing varies for different APIs. In the Java client API the ExternalClientConnection class has unsubscribe methods that take a list of topic names (or selectors) or a TopicSet.

Subscription handlers

You can assign a subscription handler to a topic. Such a handler is called whenever a client attempts to subscribe to a topic. It is called after any other authorization processes in order that it can perform topic-specific authorization for a particular client's request to subscribe to the topic. You can also delegate authorization to some asynchronous process and defer subscription until a response is received.

Topic-specific authorization

Simple authorization using a subscription handler

The following example shows a subscription handler being used to prevent mobile clients from accessing a topic:

```
TopicDefinition topicDef =
       new TopicDefinition(
           TopicDataType.SINGLE_VALUE,
           MetadataFactory.newFieldMetadata(MDataType.STRING));

       topicDef.setProperty(
           TopicProperty.SUBSCRIPTION_HANDLER,

           new TopicSubscriptionHandler() {
               @Override
               public boolean clientSubscriptionRequest(
               TopicClient client,
               Topic topic) throws AuthorisationException {
                   ConnectionType type = client.getConnectionType();
                   if (type.isCategory(ConnectionCategory.MOBILE)) {
                       throw new AuthorisationException(
                           "Mobile clients not permitted");
                   }
                   return true;
               }
           });

       addTopic("NonMobile",topicDef);
```

Delegated authorization and deferred subscription

Delegating topic authorization using a subscription handler

Another use of subscription handlers is to delegate the authorization of subscription to some asynchronous process and performing the subscription on receiving a positive response. The following example shows the use of a subscription handler to delegate the authorization of a client's subscription to an event publisher and then subscribe when the publisher gets a response.

```
   @Override
   protected void initialLoad() throws APIException {

       TopicDefinition topicDef =
       new TopicDefinition(
           TopicDataType.SINGLE_VALUE,
           MetadataFactory.newFieldMetadata(MDataType.STRING));

       topicDef.setProperty(
           TopicProperty.SUBSCRIPTION_HANDLER,
           new TopicSubscriptionHandler() {
               @Override
               public boolean clientSubscriptionRequest(
               TopicClient client,
               Topic topic) throws AuthorisationException {
                   try {
                       TopicMessage message =

theEventPublisher.createDeltaMessage("AuthTopic");
                       message.putFields(
                           client.getClientID(),
                           client.getCredentials().getUsername(),
                           client.getCredentials().getPassword(),
                           topic.getName());
                       theEventPublisher.send(message);
                   }
```

```
                    catch (Exception ex) {
                        throw new AuthorisationException(
                            "Failed to send auth request",ex);
                    }
                    return false;
                }
            });
        addTopic("Delegate",topicDef);
    }

    @Override
    protected void messageFromEventPublisher(EventConnection
 eventConnection,
    TopicMessage message) {
        if (message.getHeader(0).equals("AuthReply")) {
            try {
                Client client = Publishers.getClient(message.nextField());
                if (client!=null) {
                    Topic topic = getTopic(message.nextField());
                    if (topic!=null) {
                        if (message.nextField().equals("OK")) {

                            topic.subscribe(client);
                        }
                    }
                }
            }
            catch (Exception ex) {
                LOG.error("Error handling subscription reply",ex);
            }
        }
    }
```

The example shows the subscription handler sending a subscription request to an event publisher containing the client id, its credentials and the topic to subscribe. It then returns false so that the subscription does not happen at that point.

When an OK reply is received from the event publisher the client is subscribed. A negative reply might result in a message being sent to the client if desired as in this example the client is not aware that it had been rejected.

Note: Subscription handlers cannot be used with topics that have routing topic data.

Forced subscription

A publisher can force clients to subscribe to a topic even if the client has not requested subscription to the topic.

Individual clients

To force an individual client to subscribe to a topic, a publisher can use the `Client.subscribe` method with the force flag set to true. The following example shows a publisher that is listening for client notifications forcing all clients that connect to its topic to subscribe:

```
public void clientConnected(Client client) throws APIException {
 Topic topic = getTopic("MyTopic");
 client.subscribe(topic,true);
}
```

By topic set

The subscription can equally be done using the topic name or multiple topic names by means of a topic set as shown in the following example:

```
client.subscribe(new TopicSet("MyTopic","AnotherTopic"),true);
```

Forced subscribe all clients

You can subscribe all connected clients to a topic from within a publisher as shown in the following example:

```
Topic topic = getTopic("MyTopic");
subscribeClients(topic,true);
```

Force unsubscribing

As with subscribing, a publisher can force clients to be unsubscribed from topics. The client has methods for unsubscribing individual topics or a topic set. Using a topic set you can specify selector patterns to unsubscribe.

The following example shows a publisher unsubscribing a client from a topic on notification of its queue limit being reached:

```
public void clientQueueThresholdReached(
 Client client,
 boolean upper,
 int threshold) throws APIException {
  if (upper) {
   client.unsubscribe(getTopic("NewsTopic"));
  }
}
```

Pre-emptive subscription

Pre-emptive subscription describes the facility whereby a client can subscribe to a topic before it exists, so that if the topic does come into existence, the client is automatically subscribed to it.

All subscription and unsubscription requests that have been requested by a client are remembered in the order that they were issued. When a new topic is created the requests are applied to the topic name in order to decide whether to subscribe to that topic.

For example, if a client requested subscription to the (non-existent) topic Accounts and then used a selector to unsubscribe from all topics starting with A, when topic Accounts is created and subscribed the client does not subscribe to it.

Manual pre-emptive subscription

Publishers can use the `subscribeClients` method (with the force flag set to false) on either the publisher or topic class after creating a new topic. This causes any client that has tried to subscribe to the topic explicitly, or has subscribed with a topic selector that includes the new topic, to become subscribed to it.

The following example shows a publisher subscribing any clients that have previously registered interest in a newly created topic:

```
Topic topic = addTopic("NewTopic");
subscribeClients(topic,false);
```

Automatic pre-emptive subscription

The automatic pre-emptive subscription behavior is different between topics managed by publishers and those managed by control clients. A publisher must explicitly enable automatic pre-emptive subscription when adding a new topic for that topic and all those below it in the topic tree. When adding a topic using a control client, automatic pre-emptive subscription is enabled as default for that topic and all those below it in the topic tree.

For parts of the topic tree with automatic pre-emptive subscriptions enabled, when a topic is added, clients that have registered a pre-emptive interest in the topic are automatically subscribed.

The following example shows a publisher setting auto subscribe for a new topic. When any sub-topic is subsequently created beneath the topic, any clients that have registered an interest are automatically subscribed.

```
// Publisher example
Topic topic = addTopic("NewTopic");
topic.setAutoSubscribe(true);
```

> **Note:** The call to `setAutoSubscribe(true)` has the effect of subscribing any interested clients to the topic and any subordinates at the time of calling. A call to `subscribeClients`, as shown in the previous example, is not necessary.

Automatic pre-emptive subscription all topics

From a publisher, to set automatic subscription for all topics you can set automatic pre-emptive subscription on the whole topic tree:

```
Publishers.getTopicTree().setAutoSubscribe(true);
```

Subscribing clients individually

If required, pre-emptive subscription can also be enabled on an individual client basis using `Client.subscribe()` specifying `force=false`. There are variants of this method that take both a topic and a topic set.

Topic selection

There are a number of places where you can query the Diffusion server for multiple topics whose topic paths match a certain criteria. For example, when subscribing to a topic. To simplify the processing required, particularly when dealing with hierarchic topics, you can specify these topics using topic selectors.

Topic selectors in the Unified API

A topic selector identifies one or more topics. You can create a `TopicSelector` object by passing in a pattern expression to the `TopicSelectors parse` method.

Pattern expressions

Use pattern expressions to create a topic selector of one of the types described in the following table. The type of the topic selector is indicated by the first character of the pattern expression.

Table 46: Types of topic selector

Topic selector type	Initial character	Description
Path	>	A path pattern expression must contain a valid topic path. A PATH selector returns only the topic with the given path. See Path examples on page 144
Split-path	?	A split-path pattern expression contains a list of regular expressions separated by the / character. Each regular expression describes a level in the topic hierarchy. A SPLIT_PATH_PATTERN selector returns topics for which each regular expression matches the part of the topic path at the corresponding level. See Split-path examples on page 145
Full-path	*	A full-path pattern expression contains a regular expression. A FULL_PATH_PATTERN selector returns topics for which the regular expression matches the full topic path. See Full-path examples on page 145 **Note:** Full-path pattern topic selectors are more powerful than split-path pattern topic selectors, but are evaluated less efficiently at the server. If you are combining expressions, use selector sets instead.
Selector set	#	A selector set pattern expression contains a list of selectors separated by the separator ////. A SELECTOR_SET topic selector returns topics that match any of the selectors. **Note:** Use the `anyOf()` method for a simpler method of constructing SELECTOR_SET topic selectors. See Selector set examples on page 146

Diffusion topic selectors use Java-style regular expressions. Any Java-style regular expression can be used in split-path and full-path patterns, with the following restrictions:

- It cannot be empty
- In split-path patterns, it cannot contain the path separator (/)
- In full-path patterns, it cannot contain the selector set separator (////)

For more information about Java-style regular expressions, see Java regular expressions.

Descendant pattern qualifiers

You can modify split-path or full-path pattern expressions by appending a descendant pattern qualifier. These are described in the following table:

Table 47: Descendant pattern qualifiers

Qualifier	Behavior
None	Select only those topics that match the selector.
/	Select only the descendants of the matching topics and exclude the matching topics.
//	Select both the matching topics and their descendants.

Topic path prefixes

The topic selector capabilities in the Unified API provide methods that enable you to get the topic path prefix from a topic selector.

A topic path prefix is a concrete topic path to the most specific part of the topic tree that contains all topics that the selector can specify. For example, for the topic selector `?foo/bar/baz/.*/bing`, the topic path prefix is `foo/bar/baz`.

The topic path prefix of a selector set is the topic path prefix that is common to all topic selectors in the selector set.

Path examples

The following table contains examples of path pattern expressions:

Expression	Matches alpha/beta?	Matches alpha/beta/gamma?	Notes
>alpha/beta	yes	no	
>/alpha/beta/	yes	no	This pattern expression is equivalent to the pattern expression in the preceding row. In an absolute topic path, leading and trailing slash characters (/) are removed because the topic path is converted to canonical form. A path pattern expression can return a maximum of one topic. The trailing slash in this example is not treated as a descendant qualifier and is removed.
>alpha/beta/gamma	no	yes	
>beta	no	no	The full topic path must be specified for a path pattern expression to match a topic.
>.*/.*	no	no	For clients using the Unified API, the period (.) and asterisk (*) characters are valid in topic names. In a path pattern expression these characters match themselves and are not evaluated as part of a regular expression.

Split-path examples

The following table contains examples of split-path pattern expressions:

Expression	Matches alpha/beta?	Matches alpha/beta/gamma?	Notes
?alpha/beta	yes	no	
?alpha/beta/	no	yes	The trailing slash character (/) is treated as a descendant pattern qualifier in split-path pattern expressions. It returns descendants of the matching topics, but not the matching topics themselves.
?alpha/beta//	yes	yes	Two trailing slash characters (//) is treated as a descendant pattern qualifier in split-path pattern expressions. It returns matching topics and their descendants.
?alpha/beta/gamma	no	yes	
?beta	no	no	
?.*	no	no	Each level of a topic path must have a regular expression specified for it for a split-path pattern expression to match a topic.
?.*/.*	yes	no	
?alpha/.*//	yes	yes	In this pattern expression, "alpha/.*" matches all topics below alpha in the topic hierarchy. The descendant pattern qualifier (//) indicates that the matching topics and their descendants are to be returned.

Full-path examples

The following table contains examples of full-path pattern expressions:

Expression	Matches alpha/beta?	Matches alpha/beta/gamma?	Notes
*alpha/beta	yes	no	
*alpha/beta/gamma	no	yes	
*alpha/beta/	no	yes	The trailing slash character (/) is treated as a descendant pattern qualifier in split-path pattern expressions. It returns descendants of the matching topics, but not the matching topics themselves.
*alpha/beta//	yes	yes	Two trailing slash characters (//) is treated as a descendant pattern qualifier in split-path pattern expressions. It returns matching topics and their descendants.

Expression	Matches alpha/beta?	Matches alpha/beta/gamma?	Notes
*beta	no	no	In a full-path pattern selector the regular expression must match the full topic path for a topic to be matched.
*.*beta	yes	no	The regular expression matches the whole topic path including topic separators (/).

Selector set examples

Use the `anyOf` methods to create a SELECTOR_SET `TopicSelector` object.

The following example code shows how to use the `anyOf(TopicSelector... selectors)` method to create a selector set topic selector:

```
// Use your session to create a TopicSelectors instance
TopicSelectors selectors = session.topicSelectors();

// Create topic selectors for the individual pattern expressions
TopicSelector pathSelector = selectors.parse(">foo/bar");
TopicSelector splitPathSelector = selectors.parse("?f.*/bar\d+");
TopicSelector fullPathSelector = selectors.parse("*f.*\d+");

// Use the individual topic selectors to create the selector set topic
 selector
TopicSelector selector = selectors.anyOf(pathSelector, splitPathSelector,
 fullPathSelector);

// Use the topic selector as a parameter to methods that perform actions
 on topics or groups of topics
```

The following example code shows how to use the `anyOf(String... selectors)` method to create the same topic selector as in the previous code example, but in fewer steps:

```
// Use your session to create a TopicSelectors instance
TopicSelectors selectors = session.topicSelectors();

// Create the selector set topic selector by passing in a list of
// pattern expressions
TopicSelector selector = selectors.anyOf(">foo/bar", "?f.*/bar\d+", "*f.*
\d+");

// Use the topic selector as a parameter to methods that perform actions
 on topics or groups of topics
```

Topic selectors in the Classic API

A topic selector is a string that can be used by the Classic API to select more than one topic by indicating that subordinate topics are to be included or by fuzzy matching on topic names or both.

Including subordinate topics

When specifying a topic name you can also indicate that all of its subordinate topics are to be included by suffixing the name with a slash character (/).

For example, to select all of the subordinate topics of a topic named “MyTopic” use a selector of the format `MyTopic/`.

This notation can also be used with hierarchic topic names. So to select all topics subordinate to the topic named “A/B”, use a selector of the following format:

```
A/B/
```

Specifying a suffix of "/" does not include the topic named prior to the final "/". To include the specified topic and all of its subordinates, use "//".

For example, to select the topic "A/B" and all of its subordinates, specify the following selector:

```
A/B//
```

Fuzzy matching

A number of topics can be selected using a single topic selector which uses regular expressions to match against topic names.

Regular expressions provide a powerful mechanism for String pattern matching which is not discussed here. A Java tutorial is available for those familiar with Java or a more generic tutorial can be consulted for other language users. The regular expression syntax supported by Diffusion is defined by the Java Pattern class.

The important point to note about the use of regular expressions in topic selectors is that they are hierarchic (separated by /) and that each part of a selector can be a regular expression pattern. A multi-part selector is evaluated part by part starting from the top level of the topic tree in an attempt to find a matching topic in the tree.

If there is no "/" in the selector string but there are regex characters then the whole string is applied as a regex against the full topic name. This can be useful in some circumstances, for example when you do not know how many topic levels exist but it is far less efficient than per-level matching.

Example of selector regular expression processing

Consider the following topic selector:

```
A.*/B/.*X
```

As the above selector has three specifications it only matches with topic names with three parts.

The first check is to select any topics at the top-level of the topic tree whose name matches the pattern "A.*". In regular expression notation this pattern matches with any String that starts with "A" and is followed by zero or more other characters. So if there are no top-level topics that start with an "A" then this selector matches with no topics at all. However, for any top-level topic that does match, processing goes to the next part which is a simple String "B" and only matches with a subordinate topic called "B". So if any topics called "B" are found under the top-level topics starting with "A", processing goes onto the final pattern ".*X" which in regex notation indicates any String with any number of characters before a final "X" (a String ending with "X").

So to summarize, the above selector matches only with hierarchical topic names with 3 elements where the first element starts with an "A", the second element is "B" and the third element ends with an "X".

The following are matches:

```
Accounts/B/TAX
```

```
A/B/X
```

```
Admin/B/ProjectX
```

```
But the following are not:
```

```
Accounts/TAX
```

```
Admin/B/ProjectYHR/B/TAX
```

If you do not know how many levels you are dealing with but want to select topics where the lowest level topic name is "ABC", you must use a whole topic name selector as follows:

```
.*\x2FABC
```

The use of "\x2F" is necessary to represent a "/" as otherwise the selector evaluates per level.

So this selector matches with:

```
A/B/ABCX/Y/ZZZ/wwwwww/ABC
```

but not with:

```
A/B/CABC
```

Mixed Mode selectors

It is permitted to mix regex pattern handling and subordinate topic suffixes in the same topic selector pattern.

So it is permitted to use a selector of the form "A.*/Address/" which has the effect of selecting all of the topics subordinate to the topic named "Address" within any top level topic starting with "A".

Selector examples

Table 48: Selector examples

Selector pattern	Selects	Examples
A/	All topics subordinate to, but not including the top-level topic named "A".	A/B, A/B/C
A//	The top level topic named "A" and all topics subordinate to it.	A, A/B, A/B/C
A.*	All topics that start with the letter "A"	A, Accounts, Admin
A.*//	All topics that start with the letter "A" and their subordinates.	A, Accounts, A/B, Admin/X/Y
Counties/.*ex	All topics under the top-level topic called "Counties" whose name ends in "ex".	Counties/Middlesex, Counties/Sussex
.*folk	All topics whose full name ends in "folk" regardless of level.	UK/Counties/Suffolk, Counties/Norfolk

When is a topic string a selector?

In certain parts of the APIs only topic names can be specified (for example, when adding a topic) but in other areas selectors are allowed (for example, in subscription).

A topic string is considered to be a selector if it is terminated by a "/" or it contains any one of the regular expression metacharacters which are defined as the following characters: `[]\^$.|?*+()`

Topic loading

The act of providing the client with the current state of the topic data is called topic loading.

A Publisher maintains (or provides) one or more topics, and typically, for each topic it provides it must maintain the state of the data relating to the topic. Any client that subscribes to a topic must be provided with the initial state of the topic's data before it can start receiving updates to the topic's data.

The act of providing the client with the current state of the topic data is called topic loading. A special type of message called a topic load message is provided for this purpose. This allows the client to distinguish between a load message which represents all of the topic data and a delta message which represents an update to the topic data.

When a client subscribes to a topic, the publisher is notified through its subscription method. At this point that the publisher can provide a snapshot of the current topic state to the client (a topic load).

Subscription processing

The default implementation of the subscription method is as follows:

```
    protected void subscription(Client client,Topic topic,boolean loaded)
    throws APIException {
        if (!loaded) {
            if (topic.hasData()) {
                client.send(topic.getData().getLoadMessage(client));
            }
            else {
                LogWriter logger = getLogger();
                if (logger.isFinestLogging()) {
                    logger.finest("No action in subscription method for
 topic "
                            +topic);
                }
            }
        }
    }
```

This caters for the topic being pre-loaded by a topic loader or the topic using topic data. If there are any topics which maintain data but do not use topic data or topic loaders, the subscription method must be overridden.

Simple topic loading

The following example shows a publisher sending a client a topic load message on subscribing to a topic:

```
protected void subscription(Client client,Topic topic,boolean loaded)
throws APIException {
        TopicMessage
  loadMessage = topic.createLoadMessage();
  populateLoadMessage(loadMessage);
  client.send(loadMessage);
}
```

The above example creates a topic load message and populates it with data every time a client subscribes. Usually, it is more efficient to populate a topic load message only when the data actually changes, for example,

```
private TopicMessage theLoadMessage = null;

protected void initialLoad() throws APIException {
 updateLoadMessage("Initial data");>
}
protected synchronized void subscription(
Client client,Topic topic,boolean loaded)
throws APIException {
    client.send(theLoadMessage);
}
synchronized void updateLoadMessage(String data) throws APIException {
    theLoadMessage=createLoadMessage("MyTopic");
 theLoadMessage.put(data);
```

```
}
```

Load message

The following example shows a publisher sending topic load message to a client upon subscription:

```
protected void subscription(Client client,Topic topic,boolean loaded)
 throws APIException {
     TopicMessage loadMessage = topic.createLoadMessage();
     populateLoadMessage(loadMessage);
     client.send(loadMessage);
 }
```

Cached load message

The above example creates a topic load message and populates it with data every time a client subscribes. Usually, it is more efficient to populate a topic load message only when the data actually changes, for example:

```
private TopicMessage theLoadMessage = null;

protected void initialLoad() throws APIException {
    updateLoadMessage("Initial data");
}

protected synchronized void subscription(
Client client,Topic topic,boolean loaded)
throws APIException {
    client.send(theLoadMessage);
}

synchronized void updateLoadMessage(String data) throws APIException {
    theLoadMessage=createLoadMessage("MyTopic");
    theLoadMessage.put(data);
}
```

In the above example the updateLoadMessage method is called from elsewhere whenever the data changes. The format of the data is not addressed in this simple example.

The above examples also assume that the publisher has only one topic, which is usually not the case. Where there are multiple topics, the subscription method must return a topic load message for the topic being subscribed to. This can lead to complex 'if.. else' processing in the subscription method. This can be overcome by using either topic data subscription or topic loaders.

Topic data subscription

The recommended approach to handling topic state is using topic data. In this case the data for a topic is held with the topic. When using this pattern then subscription handling can be as simple as shown below.

```
protected void subscription(Client client,Topic topic,boolean loaded)
throws APIException {
    client.send(topic.getData().getLoadMessage());
}
```

However, as the default implementation does this anyway, you do not have to provide any processing in the subscription method.

For full details of the facilities available for handling data in this way see the section on Topic data.

Topic loaders

When multiple topics are in use (that do not use topic data), it can be inefficient to write code in the subscription method which checks which topic is being subscribed to and then behave differently. This can be overcome by specifying topic loaders for the topics.

A topic loader is an object of type TopicLoader that is declared within a publisher for a particular topic (or topics) using the addTopicLoader method. Whenever a client subscribes, if a topic loader has been defined for the topic, it is called to perform the topic load before the subscription method of the publisher is called.

Declaring a topic loader

A topic loader must normally be declared before adding the topic or topics that it loads. The following example shows a topic loader being declared for a single topic before it is created in the publishers initial load:

```
    protected void initialLoad() throws APIException {
        addTopicLoader(new MyTopicLoader(),"MyTopic");
        addTopic("MyTopic");
    }
```

Mixing simple and topic loader loading

The TopicLoader.load method returns a boolean result to indicate whether it has actually loaded the topic. This allows a TopicLoader to be employed which loads only some topics, returning false for those it does not load.

If a topic has been loaded by a topic loader, when the publisher's subscription method is called the loaded parameter is true. If loaded is false, it indicates that the subscription method must load the topic for the client.

You can use a topic loader to load some topics and allow the subscription method to load others.

Topic loaders for multiple topics

To declare a single topic loader for multiple topics, you can specify a topic selector pattern to the `addTopicLoader` method instead of a single Topic name. For example, the following declares a topic loader for the topic named “Accounts” and all subordinate to it:

```
    addTopicLoader(new MyTopicLoader(),"Accounts//");
```

Cached topic loader

The simple example shown above creates a new topic load message for every subscription. Because this is inefficient the abstract CachedTopicLoader is provided which can be extended to create a tailored topic loader which caches the topic load message. The abstract class creates the message and call `populateMessage` to set data in the message and send the message to the client. However, if called again, rather than recreating the message it uses the same one as before unless its setDirty method has been called in the interim. This makes it unsuitable for use with multiple topics. Do not add it using a topic selector that matches more than one topic.

The following example shows the use of CachedTopicLoader, declared as an anonymous class. If the `updateData` method is called, the topic loader is set as dirty and so on the next client subscription it regenerates the topic load message by calling populateMessage before sending it to the client.

```
    private CachedTopicLoader theTopicLoader = null;
    private Object theData;
    private ReentrantLock theLock = new ReentrantLock();

    protected void initialLoad() throws APIException {
        theTopicLoader =
```

```
            new CachedTopicLoader() {
                protected void populateMessage(TopicMessage message)
                throws APIException {
                    theLock.lock();
                    try {
                        message.put(theData.toString());
                    }
                    finally {
                        theLock.unlock();
                    }
                }
            };

        addTopicLoader(theTopicLoader,"MyTopic");

        addTopic("MyTopic");
    }

    void updateData(Object newData) {
        theLock.lock();
        try {
            theData=newData;
            theTopicLoader.setDirty();
        }
        finally {
            theLock.unlock();
        }
    }
```

Topic data

Diffusion topic data provides thread-safe management of data associated with a topic or special topic functionality.

A topic typically has data associated with it. A publisher is responsible for maintaining the state of the topic's data. This involves initializing the data as necessary when the publisher starts and applying any updates to the data during the time that the publisher is available.

When a client subscribes to a topic, the publisher sends the client a topic load message containing the current state of the data. Whenever the data is updated, all clients that are subscribed to the topic are sent only the differences as a delta message.

Although what is described above is the normal model for maintaining topics, Diffusion does not impose any strict rules upon how this is achieved or even whether topics must be used in this way. The data can be maintained in any form and the messages sent out can be structured in any way. The data can be maintained separately from the topic or it can be attached to the topic for ease of processing. The publisher is responsible for maintaining the integrity of the data at any time and ensuring that any new subscriber gets the latest state followed by any updates required to keep it up to date.

Even though Diffusion does not impose data handling mechanisms and message formats it does provide a feature called topic data which can greatly simplify the handling of the data associated with a topic in the following ways:

- Provides a standard way of encapsulating data in a single topicData object that can be associated with a topic. This allows for a simplified subscription process as a standard method can return the current state.
- Provides update mechanisms which automatically compare incoming data with the existing data state and generate deltas that contain only the differences. The generated deltas are sent out to all currently subscribed clients.
- Automatically maintains a cached topic load message which can be returned to new subscribers.

- Allows for messages to be formatted in various ways, such as records format or Google protocol buffers. There is also a custom data framework to support other data requirements.
- Ensures data integrity by locking data whilst it is being updated. This ensures that no updates can occur to the data whilst a client is subscribing and ensures that the newly subscribed client receives all subsequent updates.
- Provides a generic metadata modeling feature to allow message formats to be described programmatically.

Where topic data provides a data implementation for the topic it is known as publishing topic data but there are also other types of topic data that provide functionality on the topic.

The types of topic data currently supported are described in the following table:

Publishing topic data	This is the group of topic data types that provide actual data management and publishing capabilities. A number of different data formats are supported, such as single value, record format, Google protocol buffers, and Custom format. Some of these are supported by a generic metadata framework. A special type called slave topic data allows a topic to point to another topic.
Paged topic data	Facilitates a paged topic where the data for the topic is arranged into lines and each client can have a different paged view of the data where it views pages of lines, one page at a time.
Routing topic data	This is a special form of topic data that allows a topic to point at one or more other publishing topics on a client by client basis. This means that different clients can subscribe to the same (routing) topic but see different data.
Child List topic data	Maintains a list of an owning topic's child topics and notifies clients when the list is changed.

Working with topic data

Various types of data are supported. The type determines how the topic handles data (how messages are interpreted and formatted) or what special feature the topic will provide. Regardless of the type of data, certain aspects of handling topic data are generic.

Creating topic data

Topic data is created using the TopicDataFactory class, For example, topic data for use with Diffusion Record format messages is created as follows:

```
RecordTopicData topicData = TopicDataFactory.newRecordData(message);
```

Similar methods exist for different types of topic data and the TopicData subtype returned has different capabilities over and above the generic capabilities described here. See the individual sections on each data type for full details.

Associating topic data with a topic

Topic data is attached to a topic at the point when the topic is registered. If you are using topic data you **cannot** use automatic topic registration via property file entries. The following example shows topic data being added to a topic during the initialization of a publisher:

```
protected void initialLoad()
throws APIException {
```

```
    RecordTopicData topicData =
 TopicDataFactory.newRecordData(MyMetadata.get("Message1"));
    addTopic("MyTopic",topicData);
}
```

In this example it is assumed that a singleton called MyMetadata exists to serve metadata definitions. This is the recommended pattern rather than defining metadata within the publishers. Once a topic is registered, clients can subscribe to it and therefore the initial state of the topic data must have been set up (if applicable) before registering the topic.

Subscribing to topic data

Topic data automatically maintains a cached topic load message representing the current state of the topic.

This greatly simplifies subscription so that all that is required in the subscription method of a publisher is the following code:

```
protected void subscription(Client client,Topic topic,boolean loaded)
throws APIException {
     client.send(topic.getData().getLoadMessage());
}
```

The above example only works if all topics of the publisher used topic data, otherwise it is necessary to check which topic was being subscribed to first.

When topic data is in use for a topic, it is also automatically locked against any updates during the time it takes for a client to subscribe. This ensures that new deltas do not get generated between the time when the topic load message is sent to the client and the clients subscription to the topic is complete.

Message encoding

You can set the encoding to be used for all load or delta messages generated by the topic data.

Lock timeouts

If a thread tries to lock the data (for example to start an update block) and the data is already locked by another thread, the thread blocks until the lock is freed. To avoid the thread blocking indefinitely a lock timeout is employed. By default the lock timeout on data is 5 seconds but this can be changed using the setLockTimeout method on the topic. If a thread cannot acquire a lock within the timeout period, a `com.pushtechnology.diffusion.api.TimeoutException` is thrown.

Publishing topic data

Publishing topic data is the main category of topic data which provides thread-safe management of data associated with a topic, catering for updating data state, deriving differences between inputs and existing data and automatically generating deltas.

There are a number of different types of publishing topic data, each supporting different data formats. The available types are listed below and the related topics describe how to use each type in more detail.

Table 49: Publishing topic data types

Single value	This supports the simplest form of data where the data for a topic is defined as a single data item. For more information, see Single value topic data on page 159.
Record	This supports a more complex record format where the data is divided into records and/or fields which are separated by known

	delimiters within messages. For more information, see Record topic data on page 160. The format of such data is described by metadata. For more information, see Metadata on page 216.
Google protocol buffers	This supports data defined by the use of Google protocol buffers. For more information, see Protocol buffers topic data on page 161.
Custom data	This provides an interface such that any data format can be supported by writing a data handler. For more information, see Custom topic data on page 165.
Slave topic data	This is a special form of publishing data that does not have data itself but instead points to another topic that does. This allows more than one named topic to be supported by a single master topic. For more information, see Slave topic data on page 166.

The remainder of this section describes features that are common to all types of publishing topic data.

Initializing publishing topic data

Topic data must typically have its initial state set up before it is attached to a topic. The PublishingTopicData interface has an initialize(TopicMessage) method which can be called to initialize from the content of a topic message, but different types of topic data have additional initialization methods. See the various sections relating to the different topic data types for full details regarding initialization.

If the data has not been initialized at the time it is attached to a topic, it will automatically be initialized to default values as indicated by its metadata

Updating publishing topic data

Updates to topic data can occur in various ways depending upon the nature of the data and its source.

However the updates occur, one of the major features of publishing topic data is that it automatically compares the inbound data (the update) against the current data state and only generates a delta message to send to subscribed clients if there have been changes.

Also when a delta message is generated it only contains the differences between the current data state and the inbound update. Exactly how this is achieved varies for different types of topic data. Refer to the relevant sections of this manual for more detail.

All types of publishing topic data allow an update using the `update(TopicMessage)` method. This caters for the situation where data updates arrive from other Diffusion components such as event publishers. The content of the message is interpreted according to the definition of the data as known to the topic data and is expected to contain a full representation of the topic data. For more information, see Updating from delta messages on page 158.

According to the type of topic data other methods for updating the data can be available. See the relevant sections for more information on update possibilities.

Update blocks

Topic data is fully thread-safe. All updates must occur within atomic update blocks where the data is locked, so two threads cannot update the data at the same time or for a subscription to obtain the data state whilst it is being updated.

An update block is started using the `startUpdate` method and is completed using the endUpdate method. After updates have occurred, a delta message to send to subscribed clients can be generated by calling the genetrateDeltaMessage method. It can then be published using a convenience method on the data.

The changes to the data are not committed until the `endUpdate` method is called

The following example shows a publisher `messageFromEventPublisher` method updating its topic data:

```
    protected void messageFromEventPublisher(
    EventConnection eventConnection,
    TopicMessage message) {
        PublishingTopicData topicData =
getTopic(message).getPublishingData();

        try {
            topicData.startUpdate();
            if (topicData.update(message)) {

topicData.publishMessage(topicData.generateDeltaMessage());
            }
        }
        catch (APIException ex) {
            LOG.warn("Unable to process message from event publisher",ex);
        }
        finally {
            topicData.endUpdate();
        }
    }
```

Always use the try/finally pattern shown above to ensure that locks are not inadvertently left on the data.

The delta message must be generated within the update block and after all updating is complete. The delta message can be altered (for example, to set encoding) before publishing but it must be published whilst still within the update block.

The topic load message (available using the `getLoadMessage` method) is available only after the block has successfully ended (If called during the block,a load message representing the state before the block started is returned).The exception to this is if the incoming message was itself a topic load in which case the total state of the data is replaced by the update call.

> **Note:** The above example does not handle the possibility of topic loads being received from the event publisher.

Multiple updates

Any number of updates can occur within a single update block. This is useful for types that allow update at a finer level than the whole message.

You can check whether any changes were detected before sending a delta message. For example:

```
    RecordTopicData topicData = getMyData();
        topicData.startUpdate();
        try {
            topicData.update("Record1",0,"Field1","value1");
            topicData.update("Record2",0,"Field2","value2");

            if (topicData.hasChanges()) {

topicData.publishMessage(topicData.generateDeltaMessage());
            }
        }
        finally {
            topicData.endUpdate();
```

```
        }
```

Aborting an update

If any of the updates within an update block fail (throw an exception), the update block is automatically ended (locks are released) and the data is reverted to the state that it was in before the block was started. If this occurs, calling `endUpdate` has no effect.

If other exceptions can occur within the block, they must be caught and the update aborted as shown in the following example:

```
        RecordTopicData topicData = getMyData();
        topicData.startUpdate();
        try {
            topicData.update("Record1",0,"Field1","value1");

            // Other actions that might throw exceptions
            //

            topicData.update("Record2",0,"Field2","value2");

            if (topicData.hasChanges()) {

topicData.publishMessage(topicData.generateDeltaMessage());
            }
        }
        catch (Exception ex) {
            topicData.abortUpdate();
        }
        finally {
            topicData.endUpdate();
        }
```

The `abortUpdate` method might also be used to abort the update for any other reason. Aborting the update releases the locks on the data and ensures that the data is reverted to its state before the update block was started.

Simplified updating

If multiple updates are not performed and there are no special processing requirements, there is a convenience method on PublishingTopicData which starts an update, updates and publishes a delta (if required).

For example:

```
topicData.updateAndPublish(message);

// or


topicData.startUpdate();
try {
 if (topicData.update(message)) {
  topicData.publishMessage(topicData.generateDeltaMessage());
 }
}
finally {
 topicData.endUpdate();
}
```

Different types of data can have equivalent convenience methods that take other forms of input.

Simple updating example

The following example shows a very simple publisher with a single topic using protocol buffers topic data:

```
public class SamplePBPublisher extends publisher {

    private PBTopicData theData;

    protected void initialLoad() throws APIException {
        theData=TopicDataFactory.newPBData("MyMessageClass","MyMessage");
        addTopic("MyTopic",theData);
    }

    protected void messageFromEventPublisher(
    EventConnection eventConnection,
    TopicMessage message) throws APIException {
        theData.updateAndPublish(message);
    }

    protected void subscription(Client client,Topic topic,boolean loaded)
    throws APIException {
        client.send(theData.getLoadMessage());
    }
}
```

This publisher receives messages from an event publisher and automatically send out protocol buffers delta messages when differences are detected. New subscribers receive a protocol buffers topic load message representing the full current state of the topic.

Updating from delta messages

The normal update mechanism assumes that the update message contains a full representation of the topic state and that when the update message is applied an output delta containing only the changes is generated. However, in the case where the update is itself a delta (for example, when the update is received from another server through to a local subscription) there is a method on `PublishingTopicData` called `updateAndPublishFromDelta` which can apply the changes from the delta.

This method is only supported for `RecordTopicData` and if called for other types the effect is exactly the same as calling `updateAndPublish`. In some cases (such as `SingleValueTopicData`) the effect is the same anyway as partial deltas are not used.

User headers

User headers on messages are not considered part of the state of the topic. For this reason they are not considered when processing updates (or initialize) from a TopicMessage.

If you want to add user headers to any topic load message returned by the getLoadMessage method, they must first be set using setLoadHeaders.

If you want to add user headers to a delta message, pass the required headers to the call to generateDeltaMessage.

Acknowledged messages

If you want topic load messages generated by the topic data to be acknowledged, you must indicate this by calling `setLoadAckRequired(true)` before `getLoadMessage`.

> **Note:** An acknowledged topic load message is not cached as it cannot be reused – a new message is generated each time.

If you want delta messages to be acknowledged, use generateAckDeltaMessage

Single value topic data

Topic data can consist of a single item of data of a specified type. For example, String, Integer, or Decimal.

Within the Java API publishing topic data can be created that maintains the data related to a topic as a single data item. This is the simplest form of topic data that can be associated with a topic.

Items are formatted strings but special data types (such as INTEGER_STRING and DECIMAL_STRING) allow for more efficient representation and comparison of numbers.

Creating single value data

Single value topic data is created using one of the methods in TopicDataFactory, In the simplest case only the data type must be specified. For example:

```
theTopicData = TopicDataFactory.newSingleValueData(MDataType.STRING);
```

The above method can be used for any type as long as the default values are acceptable. However, to use a Decimal type with a scale other than the default, you can use:

```
theTopicData = TopicDataFactory.newSingleValueDecimalData(3);
```

Alternatively you can create a custom data item where the behavior and format of the item is delegated to a handler as follows:

```
 theTopicData =
          TopicDataFactory.newSingleValueCustomData(new
DoubleFieldHandler());
```

Whatever type the item is, the initial value is determined by the data type.

Initializing single value topic data

Single value topic data can be initialized in the generic manner alternatively it is given the default value of the given data type.

In addition, the data item can be initialized before the data is attached to a topic using a variant of the initialize method which takes any object as a parameter.

For example:

```
 theTopicData.initialise("123");
```

In the above example a String is supplied but any type can be provided as it is parsed according to the data type.

Updating single value topic data

Single value topic data can be updated in the generic manner but in addition there are some simple update methods that are specific to the data type.

```
        theTopicData.startUpdate();
        try {
            if (theTopicData.update("234")) {

theTopicData.publishMessage(theTopicData.generateDeltaMessage());
            }
        }
        finally {
            theTopicData.endUpdate();
        }
```

However, unless special handling of the message is required, it is much simpler to update and publish a delta in a single call of `updateAndPublish(Object)` which parses the incoming message and only sends a delta if a change is detected.

For example:

```
theTopicData.updateAndPublish("456");
```

Record topic data

Diffusion has the concept of record data which allows the content of a message to be represented as String data separated by field and record delimiters.

The format of the topic data (the message format) is described using metadata. This enables the topic data to compare the content of fields within messages and only generate delta messages when there are actual field differences and only populate the fields that have changed. A record format topic load message is also maintained for serving to new subscribers.

All fields within records are string format but special data types (such as INTEGER_STRING and DECIMAL_STRING) allow for more efficient representation and comparison of numbers.

The nature of record data Messages is such that all possible records and/or fields must be represented within the message and fields that have not changed are sent within delta Messages as zero length Strings. This means that a client cannot differentiate between a String field that has not changed and an empty field. You can specify a special character that is used to represent an empty field.

Record metadata

Before record topic data can be used it is necessary to describe the message format in terms of metadata. See the records section for more information on how to define metadata for use with record topic data.

An MMessage created in this way can define one or more records which make up the Message.

It is recommended that metadata definitions are encapsulated in a separate class to simplify their use – see loading metadata for more information.

Creating record topic data

Record topic data is created using the TopicDataFactory, specifying the metadata that describes the message format. For example:

```
RecordTopicData topicData =
    TopicDataFactory.newRecordData(MyMetadata.get("Message1"));
topicData.setEmptyFieldValue(Message.EMPTY_FIELD_STRING);
```

The above example shows use of the recommended value for rendering empty fields. Clients must be able to handle such a value (in this case a single character of value 0x03). If this is not done, the client is unable to distinguish between a field that had not changed and one that had been changed to a zero length string. If fields can never be empty, this is not necessary.

Initializing record topic data

Record topic data can be initialized in the generic manner. Alternatively, it is given the default value of the metadata.

In addition, the values of individual records within the data can be initialized before the data is attached to a topic using a variant of the initialize method which takes a record (or records) as a parameter.

```
Record record1 =
 new Record(MyMetadata.get("Message1").getRecord("Record1"));
record1.setField("Name","Gordon Brown");
record1.setField("AccountNumber",99999);
```

```
theTopicData.initialise(record1);
Record record2 = new
Record(MyMetadata.get("Message1").getRecord("Record2"));
record2.setField("AddressLine","10 Downing Street","London");
theTopicData.initialise(record2);
```

In the above example the records within the message have single multiplicity, however this method might also be used for setting repeating records by supplying a list of records. There is also a version of the method which allows a specific occurrence of a repeating record to be initialized.

Updating record topic data

Record topic data can be updated in the generic manner but in addition there are a number of update methods that are specific to the data type.

There are methods that allow updates at a record level in a similar way to initialize (see above) but individual field values might also be updated.

```
        theTopicData.startUpdate();
        try {
            theTopicData.update(
                "Record1",0,"AccountNumber",12345);
            theTopicData.update(
                "Record2",0,"AddressLine","England");
            if (theTopicData.hasChanges()) {

theTopicData.publishMessage(theTopicData.generateDeltaMessage());
            }
        }
        finally {
            theTopicData.endUpdate();
        }
```

The above example shows single field values being updated but the methods can equally be used to update repeating field values.

When variable repeating records or fields are encountered (which can occur only at the end of a message) if the update has the same number of entries as the current data, they are compared one by one and deltas generated only for changes in the same way as for fixed data. However, when a variable update has a different number of entries from the current data, the whole repeating series is replaced by those of the update in the delta.

You can update and publish a delta in a single call of `updateAndPublish(TopicMessage)` which parses the incoming message using the associated metadata, then only changes are sent up to the current number of entries but extra entries are sent in their entirety.

The `updateAndPublish(TopicMessage)` and `update(TopicMessage)` methods expect the input message to contain a full representation of the topic state. If the input message is itself a partial delta (for example, it is received from a subscription to an upstream master server), use the method `updateAndPublishFromDelta(TopicMessage)` instead.

Protocol buffers topic data

Diffusion supports the use of Google protocol buffers as the body of messages.

Before attempting to use protocol buffers with Diffusion, read the Google overview and understand how to define and use protocol buffers generally.

Defining a protocol buffer layout

Unlike record topic data, you cannot currently define the layout of protocol buffers using Diffusion generic metadata. However, protocol buffers have their own metadata in the form of proto files which are compiled using the `protoc` compiler to produce buffer descriptions that can be used at run time.

When using the Java API and protocol buffers topic data it is assumed that the protoc compiler has been used to generate Java classes which define the protocol buffers that are to be used.

The following is an example of a proto file that is used in examples below:

```
package testmessages;

option java_package = "com.pushtechnology.diffusion.test.data";
option java_outer_classname = "TestMessagesProto";

enum PhoneType {
  MOBILE = 0;
  HOME = 1;
  WORK = 2;
}

message PhoneNumber {
  required string number = 1;
  optional PhoneType type = 2 [default = HOME];
}

message Name {
  optional string firstname = 1;
  optional string surname = 2;
}

message StreetAddress {
  optional int32 number = 1;
  optional string street = 2;
}

message Address {
  optional StreetAddress streetAddress = 1;
  optional string town = 3;
  optional string state = 4;
}

message Person {
  optional Name name = 1;
  required Address address1 = 2;
  optional Address address2 = 3;
  repeated PhoneNumber phone = 4;
  optional string email = 5;
}
```

Creating protocol buffers topic data

Protocol buffers topic data is created using the TopicDataFactory, specifying the name of the class generated by `protoc` that contains the protocol buffer message layout and the name of the actual message within this definition.

For example, to create topic data based on the "Person" definition in the proto example shown above:

```
       PBTopicData topicData =
            TopicDataFactory.newPBData(

 "com.pushtechnology.diffusion.test.data.TestMessagesProto",
               "Person");
```

The class name must indicate a class generated by `protoc` from a proto definition. This class must be on the caller's classpath.

The message name refers to a protocol buffer message definition within the class. This is necessary because a proto definition can contain more than one message definition.

Update modes

The topic data has two modes of operation, partial and full as described below. Partial mode is the default but the mode can be changed after creating the data as follows:

```
topicData.setUpdateMode(UpdateMode.FULL);
```

Partial update mode

In this mode it is assumed that the updates to the data (see below) represent partial updates in that only the optional fields that are to change must be supplied (plus any required items). All supplied fields are compared to existing field values in the topic data and a delta message is generated with only the fields that have changed.

> **Note:** The delta also contains all required fields whether they have changed or not. Also any change to repeating fields (or messages) results in all occurrences of the repeating field being included in the delta.

Because the absence of an optional field does not indicate its removal in partial mode, to remove optional fields the field must be supplied containing the deletion value (see below). Only optional string fields can be removed in this way. The removal of optional fields of types other than string is not supported.

Full update mode

In this mode it is assumed that the updates to the data represent the full state of the data. All supplied fields are compared with the existing field values and a delta message is generated with only the fields that have changed.

> **Note:** The delta also contains all required fields whether they have changed or not. Also any change to repeating fields (or messages) results in all occurrences of the repeating field being included in the delta.

The absence of an optional field in the update indicates that the field is not required. If the base data contained a value for that field, the field is included in the delta message with a value of the deletion value (see below).

Deletion notifications

Delta messages that are sent to clients are designed to send only changes to the data to minimize the amount of data sent to the client. However, this presents a problem because there is no way within a Google protocol buffer to indicate that an optional field that previously had a value has now been removed. To overcome this the data has the concept of a deletion value which is a string that can be sent as the content of a field to clients to indicate that the field has been removed. The deletion value can be a String of any length and a value that does not occur as a valid value in any fields is chosen. The value can be set as follows after creating the data:

```
topicData.setDeletionValue("$X$");
```

If no value is explicitly specified, a value of `<DEL>` is assumed.

When operating in partial update mode, this value can also be supplied as the value of a field in the update message to indicate the removal of the field.

Deletion notifications apply only to fields of type string. The notification of deletion of all other field types is not supported.

Initializing protocol buffers topic data

Protocol buffers topic data can be initialized in the generic manner. If the topic data is explicitly initialized, it is initialized to the default values as indicated by the proto definition.

Alternatively the data can be initialized using a variant of the initialize method which takes a Google protocol buffers AbstractMessage as a parameter.

The following example shows a Google protocol buffers message of type Person being built and used to initialize the data:

```
Person.Builder person = Person.newBuilder();

person.setName(
 Name.newBuilder().
  setFirstname("Gordon").
  setSurname("Brown"));

person.setAddress1(
 Address.newBuilder().setStreetAddress(
        StreetAddress.newBuilder().
   setNumber(10).
   setStreet("Downing Street")).
         setTown("Westminster").
         setState("London"));

person.addPhone(
    PhoneNumber.newBuilder().
        setNumber("071 2345678").
        setType(PhoneType.HOME));

topicData.initialise(person.build());
```

Updating protocol buffers topic data

Protocol buffers topic data can be updated in the generic manner but alternatively it can be updated from a protocol buffers message of the same type as the data.

For example:

```
   void updateData(PBTopicData topicData,Person person) throws
APIException {
       topicData.startUpdate();
       try {
           if (topicData.update(person)) {

topicData.publishMessage(topicData.generateDeltaMessage());
           }
       }
       finally {
           topicData.endUpdate();
       }
   }
```

You can update and publish a delta in a single call of `updateAndPublish(TopicMessage)`, which parses the incoming message using the protocol buffers message description. Alternatively you can use `updateAndPublish(AbstractMessage)` which is the equivalent to calling `update(AbstractMessage)` to perform the updating.

The data that is input on an update is checked against the current data state and a delta protocol buffer with only those fields that differ are generated. If there are no differences, no delta is required. If multiple updates are done within the same block, the update effect is cumulative in that the newly generated protocol buffers delta is merged with any existing delta. When repeating fields are used it is important to note that protocol buffers merge repeating fields by appending them. When `generateDeltaMessage` is called, the message is populated with the cumulative protocol buffers delta.

> **Note:** Required fields are always sent in deltas. To minimize message size their use is discouraged. Also, if a repeated field (or message) is changed, it is necessary to transmit all entries again.

Recommendations for usage

Because there are some limitations to the use of Google protocol buffers, if possible, observe the following recommendations when creating the layout:

1. Avoid the use of required fields (or messages) as their values must always be transmitted in deltas. Google recommends, all fields be optional.
2. Avoid the use of optional field types other than string as the notification of the removal of such fields is not supported.
3. Avoid the use of repeated fields or messages as changes to these always require the transmission of all field occurrences in the delta.

Using protocol buffers within clients

Within a client application there is no Diffusion specific API for handling protocol buffers. Instead the user serializes the protocol buffer into the message.

For example, in the Java API:

```
void writePersonToMessage(Person person,TopicMessage topicMessage)
throws IOException {
     person.build().writeTo(topicMessage.getOutputStream());
}
```

Reading a protocol buffer from a Diffusion message is equally straightforward in Java:

```
Person readPersonFromMessage(TopicMessage topicMessage)
throws IOException {
    return Person.parseFrom(topicMessage.getInputStream());
}
```

When using client applications in other languages, the protocol buffers API is be different. See the various protocol buffers API implementations for details on each supported language.

Custom topic data

You can define your own formatting, structure, and behavior for topic data by writing an implementation of `PublishingTopicData`. This allows for data types and layouts that are not supported directly by Diffusion.

When custom topic data is used a user-created data handler class is nominated which is responsible for maintaining the data, performing comparisons on the data and populating topic load and delta messages.

Writing a custom topic data handler

A custom topic data handler handles the state of the data for an instance of CustomTopicData.

The handler must implement the `CustomTopicDataHandler` interface. The methods on this interface are called by the custom topic data implementation. Do not call them directly. A handler can also extend `AbstractCustomTopicDataHandler` which provides default implementations of optional methods.

A handler can initialize the data state in any way that is suitable. The initialization of the data can occur when the handler is created or as a result of `initialize(TopicMessage)` being called on the topic data which in turn invokes `initialize(TopicMessage)` on the handler.

When the data is associated with a topic `prepare()` is called to allow for initialization to be performed (if not done already).

The data can be updated through messages in the normal way by using `update(TopicMessage)` on the topic data which in turn calls `update(TopicMessage)` on the handler.

Alternatively the data can be updated in some other way, but if this is the case then it is important that it is done within update blocks. If errors occur within updating, report this to the topic data using `errorsInUpdate()` on the topic data.

When an update block is started `startUpdate()` is called to allow the handler to prepare for updating

If the update is aborted, `abortUpdate()` is called to allow the handler to discard updates.

However the data is updated it is important that the actual state of the data is not updated until `endUpdate()` is called.

The `populateDelta(TopicMessage)` method can be called within an update block to write the delta to the message. This is called only if `hasChanges()` returns true.

The `populateTopicLoad(TopicMessage)` is called when a topic load is required and it writes the current state to the message.

Note: Headers or ack flags are set (if required) before the populate methods are called.

Creating custom topic data

Custom topic data is created using the TopicDataFactory, specifying the custom topic data handler to use.

For example:

```
CustomTopicData topicData =
       TopicDataFactory.newCustomData(
           new MyHandler());
```

Slave topic data

Slave topic data acts as an alias to another topic. It does not contain any data itself, but instead points to another topic that does. More that one slave topic can point to the same master topic. The master topic cannot also be a slave topic.

Creating slave topic data

Slave topic data creation is shown in the following example:

```
// create metadata
MMessage theMetadata =
 MetadataFactory.newMetadata("slave",TopicDataType.RECORD);
MRecord record = theMetadata.addRecord("data");
MField field = record.addField("value", MDataType.STRING);
field.setDefaultValue("example");

// Create topic data
RecordTopicData topicData = TopicDataFactory.newRecordData(theMetadata);
addTopic("MyTopic", topicData);

SlaveTopicData slaveData = TopicDataFactory.newSlaveData(topicData);
master.addTopic("MySlave",slaveData);
```

This shows a slave topic being defined which points to a master topic with record topic data. In this example any client that subscribes to MySlave gets the state of MyTopic and any updates to it. Updates can be applied to MySlave and the effect is to update MyTopic and propagate the changes to any other slaves of MyTopic.

Initializing slave topic data

Slave topic data does not have to be initialized as it is the master topic that contains the data. However, if the initialize method is called on a slave the effect is to call the master's initialize method.

Effects of updates to the master topic

If a topic has publishing topic data which is pointed to by one or more instances of slave topic data, any updates to it are propagated to the slaves. This means that clients subscribed to the slave topics receive deltas of changes to the master topic.

Effect of updates to a slave topic

It is also permitted for updates to be performed to slave topic data as if it were the actual topic data. However, all of the type-specific update methods are not available. When a slave is updated, the master topic is updated and deltas are sent out to clients subscribed to the master and all slaves.

Effect of removing the master topic

If a master topic is removed that has slave topics pointing at it, all slave topics are also removed.

Paged topic data

Paged topic data provides the functionality of a paged topic. The data is formatted in lines. A client can view the data as pages made up of one or more lines and can page forward and backward through the data.

Note: Paged topic data is not supported by the Introspector or console.

Paged topic data provides you with the following capabilities:

- Use a publisher or control client to manage the data associated with a topic as lines, where there can be any number of lines of data associated with the topic and lines can be added, updated, or removed.
- Use a publisher or control client to organize the lines in an order specified by a user-supplied comparator.
- Use a subscribing client to open a view on the data that indicates how many lines of data the client views per page.
- Use a client to page through the data, selecting next, prior, first, last, or a numbered page.
- Receive notifications in the subscribing client if a change in some way makes its current page invalid. For example, if lines have been added prior to the page making its pagination invalid or if data has changed on the current page. Clients can refresh the page if required.
- Receive notifications in the subscribing client of updates to lines on its current page.
- Receive notifications in the subscribing client of additions to its current page if it is on the last page and has room for more lines.

Table 50: Paged topic data types

String	Where each line is a Java String (variable length text).
Record	Where each line is a record comprising a variable number of fields that can be of different data types.

Managing paged topic data in the Unified API

You can use the Unified API to create and update paged topics.

Creating paged topics

Use the TopicControl feature to create paged topics.

Updating paged topics

Use the TopicUpdateControl feature to update paged topics.

The TopicUpdateControl feature provides factory interfaces that can be used to create updates for paged topics.

Related Links

[Creating paged topics](#) on page 231
Use the `TopicDetails.Builder` classes with the TopicControl feature to create paged topics.

[Building updates for paged topics](#) on page 235
Use the `UpdateFactory` classes with the TopicUpdateControl feature to create the `Update` objects that can be used to update paged topics.

Managing paged topic data in the Publisher API

You can use the Publisher API to create and update paged topics.

Creating paged topic data

Create paged topic data as a String by using the following code:

```
PagedStringTopicData psData = TopicDataFactory.newPagedStringData();
```

Create paged topic data as a record (with metadata) by using the following code:

```
MRecord recordMetadata =
            MetadataFactory.newRecordMetadata("RecMetadata");
        recordMetadata.addField("Currency");
        recordMetadata.addField("Price",MDataType.DECIMAL_STRING);

        PagedRecordTopicData prData =
            TopicDataFactory.newPagedRecordData(recordMetadata);
```

In the preceding example, each line of the data conforms to a simple metadata definition. This is the recommended use, but you can specify the metadata as null, in which case no metadata rules are applied to the lines of data.

Creating ordered paged topic data

You can create paged topic data that is maintained in an order specified by a comparator supplied by the user. The comparator is specified when the topic data is created, for example:

```
        // String type
        PagedStringTopicData psData =
            TopicDataFactory.newPagedStringData(
                new Comparator<String>() {
                    @Override
                    public int compare(String o1,String o2) {
                        return (o1.compareTo(o2)*-1);
                    }
                });
```

The comparator in the preceding example causes the lines to be maintained in reverse alphabetical order.

> **Note:** When using ordered data it is also necessary to indicate must be done with lines that are evaluated as being equal by the comparator (duplicates). The following table describes the options for handling duplicates.

Table 51: Duplicates policies

Policy	Usage
`NOT_ALLOWED`	Duplicates are not allowed. An attempt to add a line that is equal to another line causes an exception. This is the default policy.
`FIRST`	If a line is being added that is equal to one or more existing lines, it is inserted before the existing line or lines.

Policy	Usage
`LAST`	If a line is being added that is equal to one or more existing lines, it is inserted after the existing line or lines.

Unless the default of `NOT_ALLOWED` is required, the policy can be set after creating the topic data as by using the following code:

```
psData.setDuplicatesPolicy(Duplicates.FIRST);
```

Initializing paged topic data

There are no specific methods for initializing the data before adding it to the topic but if an initial state is required, any of the updating methods described below can be used before the data is added to the topic.

Updating paged topic data

Paged topic data can be updated at any time using a set of methods that enable additions, updates and deletions.

All of these methods lock the data, perform the update, notify any affected clients of changes as appropriate and unlock the data. If it is required to lock the data over more than one update so that it cannot be changed whilst manipulating it, use the lock() mechanism on the data.

The method signatures vary according to the type so the line referred to in the table below refers to a String or a record as appropriate.

Some methods are usable only with unordered topic data, some only with ordered, and some behave differently according to the type.

Methods for use with ordered topic data typically act on only one record at a time and do not require an index reference to the record.

Table 52: Usable methods with ordered topic data

`add(line)`	Add the specified line to the end of the data (if unordered) or at the appropriate position (if ordered).
`add(List<line>)`	Add the specified list of lines to the data. For unordered data the lines are all added at the end. For ordered data this is the same as calling add(line) repeatedly.
`add(int,List<line>)`	Add the specified list of lines into the data at the specified index.
`update(line)`	This is functionally equivalent to calling remove(line) followed by add(line).
`remove(int,int)`	Removes one or more lines. The first number specified is the index to start from and the second is the number of lines to remove The code `remove(0,1)` removes the first line.
`remove(line)`	The current line that is equal to the specified line according to the comparator is removed. If there is more than one matching line, the duplicates policy specifies which is removed. If no matching line is found, this has no effect.

Methods for use with unordered topic data typically requires an index reference to the record that they act on.

Table 53: Usable methods with unordered topic data

`add(line)`	Add the specified line to the end of the data (if unordered) or at the appropriate position (if ordered).
`add(List<line>)`	Add the specified list of lines to the data. For unordered data the lines are all added at the end. For ordered data this is the same as calling add(line) repeatedly.
`add(int,line)`	Add the specified line into the data at the specified index. The line is inserted at the specified index. Indexing starts at 0. Using add(0,line) is the same as inserting the line at the start of the data.
`add(int,List<line>)`	Add the specified list of lines into the data at the specified index.
`update(int,line)`	Update the line at the specified index with the specified line. This effectively replaces the line with the one supplied.
`remove(int,int)`	Removes one or more lines. The first number specified is the index to start from and the second is the number of lines to remove The code `remove(0,1)` removes the first line.

Methods are also available to get specified lines or a range of lines which might help in updating.

Subscribing to paged topic data

You can use the Classic API to view paged topic data.

Client handling of paged topic data

The client application must be able to handle the Paged topic protocol. Most client APIs provide the capability for handling such topics transparently. This section shows how it is handled in the Java API.

Handling a topic load from a paged topic and opening a paged topic

A client receives messages on its listener methods and can detect a load message from a paged topic by means of the `isPagedLoad()` method. On receipt of a paged load message the client application must create a PagedTopicHandler to handle the topic. This handler provides the capability to send requests to the topic and also routes notifications from the topic to a specified PagedTopicListener. Such a handler can be created using the `ExternalClientConnection.createPagedTopicHandler` method.

The returned handler is of type `PagedTopicHandler` and the above example assumes the calling class implements `PagedTopicListener` and processes notifications.

Having created such a handler no further messages are received for that topic on the `messageFromServer` method because they are all consumed by the handler.

Before sending any commands to the topic or receiving any notifications, the client must open the topic to establish its view on the topic data.

Having declared a `PagedTopicHandler` for a paged topic all interactions with the topic use the handler. Before anything else can happen, the client application must open the topic by invoking the `open` method.

When opening, the number of lines per page and the initial page to be returned must be specified.

The number of lines per page can be any number greater than 0. It determines the size of the page that is sent to the client whenever a page is requested. The lines of the data are paginated for the client according to the number of lines per page requested.

The initial page can be any number greater than 0 for an absolute page number or -1 to indicate the current highest page. If the number specified is greater than the current highest page, the highest page is returned.

Having opened the topic, the client application receives the initial requested page on the page method of the `PagedTopicListener`.

The following code sample shows how to create a suitable topic handler on receipt of a paged load message and open the topic for pages with 10 lines per page. The highest page is returned:

```
    public void messageFromServer(
    ServerConnection serverConnection,
    TopicMessage message) {
        if (message.isPagedLoad()) {
            try {

 theHandler=theConnection.createPagedTopicHandler(message,this);
                theHandler.open(10,-1)
            }
            catch (Exception ex) {
                ex.printStackTrace();
            }
        }
    }
```

Paged topic commands

The PagedTopicHandler has the following command methods with which it can send messages to the paged topic:

Table 54: Paged topic commands

`open(linesPerPage,initialPage)`	Open the topic. The `linesPerPage` value specifies the number of lines that the client wants to view per page. The `initialPage` value specifies the first page to be returned as a page notification.
`page(PageOption)`	Request a page. The page is returned as a page notification. `PageOption` is an enum with the following possible values: • REFRESH- sends a newly formatted version of the current page. • NEXT- returns the next page (or the same page if there are no more pages). • PRIOR- returns the previous page (or the same page if on first). • FIRST- returns the first page. • LAST- returns the last (or highest) page.
`page(pageNumber)`	Request a page by absolute page number. The page is returned as a page notification. You can specify -1 to

	request the highest page. If the page number is higher than the number of pages, the highest page is returned.
`close()`	Close the topic. No more notifications are received after this is called. The topic can be opened again at some future point. This does not unsubscribe the client from the topic.

The page and close commands can be invoked only after the topic has been opened.

Client notifications on Update

Whenever an update occurs, all clients affected by the update are notified as appropriate as described in the following sections:

Additions (at end)

Whenever one or more lines are added to the end of the data, any client that is currently positioned on the last page is notified of the new lines so that it can add the lines to its page if required.

Insertions (additions at index or ordered adds)

Whenever lines are inserted, any client that is positioned on a page that contains or is after the new lines is notified that its current pagination is invalid (page is dirty). The client can choose to mark the page as dirty and refresh it on demand.

Updates

Whenever a line is updated, any client that has the line on its current page is notified of the new line value so that it can update it if required.

> **Note:** An ordered update can reposition the lines and has the effect of a remove followed by an add.

Removals

Removals are treated like insertions. Clients on pages on or after the removal point get a notification that their page is dirty.

Paged topic notifications

The `PagedTopicListener` interface has the following methods upon which it receives notifications from the paged topic:

Table 55: Paged topic notifications

`page()PagedTopicHandler,PageStatus,Lines)`	A page is returned in response to the open or to a page command. The `PageStatus` indicates the page number and the total number of pages. The status is never dirty because this is a fresh page. The `Lines` object contains all of the lines on the page. There are normally as many lines as the declared number of lines per page except if it is the last page in which case there might be less.
`statusChanged(PagedTopicHandler,PageStatus)`	The status has changed in relation to the page that the client has. This might have happened because the pagination that the client currently knows has changed or it

	might be because the client sees the page it has as being dirty because lines have been added or removed before (or on) it.
`add(PagedTopicHandler,PageStatus,Lines)`	One or more lines have been added to the end of the page that the client currently has. The client can add the lines to its view of the page. If the client does not add the lines to its view of the page, it must consider the page to be dirty.
`update(PagedTopicHandler,PageStatus,index,Lines)`	A line on the current page has been updated. The index indicates the relative index of the line within the page (where the first line is 0). The `Lines` object contains a single line of data, which can be used to replace the line in the client's view. If the client does not replace the line, it must consider the page to be dirty.

Page status

The PageStatus that is provided with all notifications is an object that contains the information described in the following table:

Table 56: Page status

CurrentPage	This indicates the current page number. When a page is being supplied as a result of a command, this is the page number by which it is known to the client. When part of a statusChangedNotification, it is the page number of the page currently known to the client which might have changed because of re-pagination if the page has become dirty.
LastPage	This is the page number of the currently known highest page.
TotalNumberOfLines	This is the current total number of lines available.
Dirty	This is true if the current page that the client has is now to be seen as out of date. This might be because of the addition or removal of lines on or before the current page. In response to this the client can choose to refresh the page if required.

Lines

The `Lines` object that is passed with some notifications encapsulates the lines of data as either Strings or records depending upon the type of the paged topic data. There are methods to determine which type the lines are and obtain them as either Strings or records.

Closing a paged topic

Issuing a close command to the paged topic by using the PagedTopicHandler stops all notifications to the client. The close command does not unsubscribe the client from the topic and the topic can be reopened at some future point.

Unsubscribing from a paged topic

If a client unsubscribes from a paged topic, must discard any handler in use. If the client wants to re-subscribe to the paged topic, it can, but the client must re-establish the handler.

Routing topic data

This is a special form of topic data that allows a topic to point at one or more other publishing topics on a client by client basis. This means that different clients can subscribe to the same (routing) topic but see different data.

The routing (by client) is determined by a special user written subscription handler that authorizes a client to use the topic and assigns a real publishing topic to represent it. The client receives the topic state (the topic load message) from the topic that it is routed to. Updates to a publishing topic linked to in such a way are propagated to clients routed to it on the routing topic.

Creating routing topic data

Routing topic data is created as follows:

```
RoutingTopicData topicData =
      TopicDataFactory.newRoutingData(new MySubscriptionHandler());
addTopic("MyTopic",topicData);
```

The subscription handler (MySubscriptionHandler) must be an implementation of the RoutingTopicDataSubscriptionHandler interface which provides the topic routing logic (see below).

The routing topic data subscription handler

When routing topic data is created it must have a subscription handler associated with it which is responsible for providing the topic routing logic.

The handler returns the actual topic that the client maps to or it might return null to indicate the authorization and mapping of the topic is to be delegated to some asynchronous process.

The client is not actually subscribed to the routing topic until the handler returns an actual topic. In the case where the handler returns null, subscription does not occur until some subsequent process calls the subscribe(TopicClient,Topic) method to provide the mapping.

Synchronous topic Mapping

The handler can be written to perform any authorization required and immediately return a topic to map to.

The following example shows a handler that authorizes the subscription using a local class and returns a different topic for mobile clients than for all other types of Client:

```
private class MySubscriptionHandler
   implements RoutingTopicDataSubscriptionHandler {
       @Override
       public Topic clientSubscriptionRequest(
       TopicClient client,RoutingTopicData topicData)
       throws AuthorisationException {
           if
(MyAuthorisationHandler.canSubscribe(client,topicData.getTopic())) {
               if (client.getConnectionType().isCategory(
                   ConnectionCategory.MOBILE))
               {
                   return theMobileTopic;
               }
               else {
                   return theNormalTopic;
               }
           }
           else {
               throw new AuthorisationException("Not authorized");
           }
       }
   }
```

Asynchronous topic mapping

As an alternative mode of operation the subscription handler can delegate the authorization and mapping of the topic to some asynchronous process.

The following example shows a handler that delegates the subscription to an event publisher and when a reply is received from the event publisher, the mapping and subscription occurs.

The subscription handler shown below sends a message to the event publisher with the client id, its credentials and the topic it is trying to subscribe to. The fact that it returns null means that the subscription will not occur at this point in time.

```
private class MyAsyncSubscriptionHandler
   implements RoutingTopicDataSubscriptionHandler {
       @Override
       public Topic clientSubscriptionRequest(
       TopicClient client,RoutingTopicData topicData)
       throws AuthorisationException {
           EventConnection eventPublisher =
               Publishers.getEventPublisher("Authorizer");
           if (eventPublisher!=null) {
               try {
                   TopicMessage message =
                       Publishers.createDeltaMessage("AuthTopic");
                   message.putFields(
                       client.getClientID(),
                       client.getCredentials().getUsername(),
                       client.getCredentials().getPassword(),
                       topicData.getTopic().getName());
                   eventPublisher.send(message);
                   return null;
               }
               catch (Exception ex) {
                   throw new AuthorisationException(
                       "Failed to send auth request",
                       ex);
               }
           }
           throw new AuthorisationException("Authorizer is not running");
       }
   }
```

The following `messageFromEventPublisher` method handles the reply which indicates if the subscription is OK and return the client ID and the topic that it must map to. This calls back on the routing topic data to do the subscription passing the mapped topic.

```
   protected void messageFromEventPublisher(
   EventConnection eventConnection,
   TopicMessage message) {
       if (eventConnection.getId().equals("Authorizer")) {
           String result = message.getHeader(0);
           if (result.equals("OK")) {
               String clientID = message.getHeader(1);
               String topicName = message.getHeader(2);
               Client client = Publishers.getClient(clientID);
               if (client!=null) {
                   Topic topic = getTopic(topicName);
                   if (topic!=null) {
                       RoutingTopicData topicData =

(RoutingTopicData)getTopic("MyTopic").getData();
                       try {
                           topicData.subscribe(client,topic);
                       }
                       catch (Exception ex) {
                           LOG.warn("Subscription failed",ex);
```

```
                                }
                            }
                        }
                    }
                }
            }
```

Client subscription

When a client subscribes to a routing topic then the subscription request is intercepted and passed to the routing topic's subscription handler (see above). The handler can return a topic to map to in which case the client immediately becomes subscribed to the routing topic. Alternatively the handler might delegate the authorization/mapping in which case the client is not subscribed but can be subscribe later by a callback on the routing topic data.

On subscription the `Publisher.subscription` method is called and the topic load message can be obtained from the routing data, but it must be obtained using the `getLoadMessage(client)` method rather than the no arguments variant as the load message varies per client. The default implementation does this anyway.

Messages published to clients

Whenever a message is published on a topic that is mapped to by a routing topic then a message is also published to all clients that map to that topic but the topic name in the message has been altered accordingly.

Messages sent from clients

Any messages sent from clients on a routing topic are routed to the publisher in the normal way.

Effect of removal of a target topic

If a topic is removed that has been mapped to by a routing topic, all clients that have been mapped to that topic are unsubscribed from the routing topic.

Child list topic data

This functional data maintains a list of an owning topic's child topics and notifies clients when the list is changed.

Within the Java API topic data can be created which automatically maintains a list of the child topics of the owning topic and sends out delta messages when a child is added or removed.

Child list topic data can be created by using the following code:

```
theTopicData = TopicDataFactory.newChildListData();
```

Topic notify topic data

This functional data enables notifications of topic creation to be sent to clients.

This is a special form of topic data that allows clients to be notified when a new topic is added. This is especially useful for publisher clients which might want to replicate topics from the master server.

A client can select which part of the topic tree it wants to receive notifications for and also the level of notification it requires. For example, it can get notification of only the topic name, or it can also request the topic metadata (where there is some) or it can even request the full definition of the topic which allows the topic to be replicated (useful mainly in publisher clients).

Creating notify topic data

Topic notify topic data is created as follows:

```
        TopicNotifyTopicData topicData =
            TopicDataFactory.newTopicNotifyData(false);

        addTopic("TopicNotifier",topicData);
```

The boolean parameter that is specified on creation indicates whether to use metadata caching. This can minimize the amount of metadata that is sent to a client by sending a named metadata definition only once and thereafter sending only its name. However, this works only if the top-level names of all metadata used in all topic definitions are unique as it is the top-level name that is used to notify clients. If the uniqueness of metadata names cannot be guaranteed, do not use metadata caching as unpredictable results can occur.

Using notify topic data

From the server/Publisher point of view once a topic notify topic is created there is no further processing required. Once the topic is available clients can subscribe to it and request notifications.

Client handling of notify topic data

How a client handles a topic notify topic

At the client end the client application must be able to handle the notify topic protocol. Some client APIs provide the capability for handling such topics transparently. This section shows how it is handled in the Java client API.

Handling a topic load

A client receives messages on its listener methods and can detect a load message from a topic notify topic by means of the `isTopicNotifyLoad()` method. On receipt of such a load message the client application must create a `TopicNotifyTopicHandler` to handle the topic. This handler will provide the facility to select the range and level of notifications and also will route notifications to a specified `TopicNotifyTopicListener`. Such a handler can be created using the `ServerConnection.createTopicNotifyTopicHandler` method.

The following code sample shows how to create a suitable topic handler on receipt of a topic notify load message and set it to received full topic add notifications:

```
    public void messageFromServer(
    ServerConnection serverConnection,
    TopicMessage message) {
        if (message.isTopicNotifyLoad()) {
            try {
                theHandler=theConnection.createTopicNotifyTopicHandler(
                    message,
                    this);

  theHandler.setNotificationDetails(NotificationLevel.FULL,false,false);
            }
            catch (Exception ex) {
                ex.printStackTrace();
            }
        }
    }
```

The returned handler is of type `TopicNotifyTopicHandler` and the above example assumes the calling class implements `TopicNotifyTopicListener` and processes notifications of topics added.

Having created such a handler no further messages is received for that topic on the `messageFromServer` method as they are all consumed by the handler.

You can request notifications of any of the following:

- Topic Additions – notification of a topic being created.
- Topic Deletions – notifications of a topic being removed.
- Topic Updates – notifications of changes to non-static topic properties. At this point in time only the topic reference is catered for.

Add notification levels

When a handler is created the required notification level for topic add notifications must be specified. The level indicates the amount of information that is notified to the client for each (selected) topic that is added. The amount of information is normally set to the least required. Full information is normally required only when replicating topics (in a publisher client).

The possible notification levels are as follows:

Table 57: Notification levels

Level	Description
MINIMUM	Only the topic name and type are notified. This is the default if not explicitly set.
PROPERTIES	The topic name, type and all topic properties that are set are notified. The properties are all of the configurable settings of the topic other than its metadata.
METADATA	The topic name, type, and metadata (for topic types that have metadata) is notified.
FULL	All details are notified – name, type, properties and metadata.
NONE	This can be used to indicate that add notifications are not required. This can be the case if the client only wants to receive delete and/or update notifications.

The notification level can be changed at any point whilst the client is subscribed to the topic (using the `setNotificationDetails` method on the handler) but it is recommended that the level is chosen at the beginning an retained for the duration of the subscription.

Selection of topics to notify

Creation of the handler does not cause the client to start receiving any notifications of topics added. The range of topics for which notifications are required must be set using the `select` method on the handler. The range is specified as a set of topic names and/or topic selector patterns. After the server receives the selection, if any topics that exist at the server match the selections, the client is notified immediately and if any topics are created that match the selections in the future (whilst the client is subscribed to the notifying topic), the client is notified at the time of creation.

Notifications of new topics always occurs before the client receives a topic load from newly added topic.

The set of selections is specified as a TopicSet and each time select is called it can be used to add to or update the current selection at the server. Selection modes for the specified set are:

Table 58: Selection modes

Mode	Description
ADD	The set of topic selectors are added to the set of selections at the server.
REPLACE	The set of topic selectors totally replaces the set of selections at the server.

Mode	Description
REMOVE	The set of topic selectors are removed from selections at the server. Only if the selector pattern is an exact match for a previously supplied pattern is the topic selector removed.
CLEAR	Removes all selections at the server.

Note: This is an asynchronous operation with the server and notifications relating to the previous selections can continue to be received by the client until the update is successfully processed by the server.

For example, the following code can be used to select a set of notifications:

```
theHandler.select(SelectionMode.ADD,new
 TopicSet("Accounts//","Trading//"));
```

The above causes notifications for all topics created under the top-level "Accounts" and "Trading" topics.

Handling notifications

All notifications of topics added are passed to the `TopicNotifyTopicListener` object specified when the `TopicNotifyTopicHandler` was created.

Topic adds

Notifications of topic added are passed on the `topicAdded` method which has two parameters as follows:

`String topicName`	The full name of the topic that has been added
`TopicDefinition definition`	The encapsulated details of the topic. Depending upon the notification level selected some details might not be available. For example, if metadata notification was not selected, the `getMetadata` method returns null. The topic type available as a `TopicDataType` – see API documentation of this for full details of all of the topic types that can occur. The topic properties are provided as a Map of key value pairs. The key is an instance of `TopicProperty` – see the API documentation of this for full details of all of the properties that can be returned. If properties are not selected, the returned Map is empty. The topic metadata is returned as an instance of MNode where appropriate. If the topic type does not have metadata or metadata notifications were not selected, this is be null.

The following example of a `topicAdded` implementation shows type, metadata and a single property being extracted from the definition passed:

```
public void topicAdded(String topicName,TopicDefinition definition) {
 TopicDataType type = definition.getType();
 MNode metadata = definition.getMetadata();
 try {
  int capacity =
 definition.getIntegerProperty(TopicProperty.DELTA_MESSAGE_CAPACITY);
 }
 catch (APIException ex) {
  ex.printStackTrace();
 }
```

```
}
```

Topic removals

Notifications of topic removals are passed on the `topicRemoved` method which simply passes the topic name.

Topic updates

Notifications of topic updates are passed on the `topicUpdated` method which passes the topic name and a map of properties that have changed. At this point in time the only property that is catered for is the topic reference (keyed by `TopicProperty.REFERENCE`).

Unsubscribing

When the client application unsubscribes from the topic notify topic, the handler will become unusable and any outstanding notifications will be discarded.

Topic fetch

A client can fetch the current state (represented as a topic load message) at any point, even if the client is not subscribed to the topic. The client can even fetch the state of a topic that does not exist (allowing for a simple request/response mechanism).

This feature works out of the box for topics that use topic data but for other topics, you must provide the handling of a fetch in the publisher.

Client fetch

How to fetch the state of a topic from a client

The client requests this using a fetch command on the API.

For example in the Java API this is done using the `fetch` method which can be used for a single topic or for a set of topics.

Topic selectors can be specified to the fetch request in which case a fetch is performed for all matching topics.

You can also provide one or more headers with the fetch request which will be reflected back in the reply for correlation purposes. If the reply message already has user headers, the request headers are be appended to the end of the header list.

The fetch request is an asynchronous request and the reply (the topic state) is returned to the client on its normal listener interface. In the Java client this is on the usual `messageFromServer` method and messages that are fetch replies can be distinguished by means of the `isFetchReply` method on the incoming message.

Server fetch handling

What happens at the server when a fetch request is received from a client

When a request to fetch an existing topic is received by the server, one of the following is done to obtain the topic state.

- The client authorization handler is called to determined whether the client can fetch the topic state. There are separate methods on this handler for dealing with normal topic names and topic selectors. If the handler rejects the fetch, no further action is taken.
- If a "fetch handler" (see below) exists for a topic, it is called to obtain the topic state. The handler can return null to indicate that the fetch is not allowed or that it is to be handled asynchronously.
- If the topic has topic data (and there was no fetch handler), the current state is obtained from the topic data.

- If there was no fetch handler and no topic data, as a last resort the fetch is delegated to the `fetchForClient` method in the publisher.

Fetch for non-existent topics

When a request to fetch the state of a topic that does not exist is received, it is reported as an event through the `ClientListener` interface on its `clientFetchInvalid` method. This routine sends a reply to the client through the `Client.sendFetchReply` method, which allows a fetch to even be used for a topic that does not exist.

Fetch batching

As a client can send a single fetch request that selects many topics, there might be a problem with queuing too many fetch replies at once for a client as the client queue might fill and the client get disconnected.

To avoid this, when a single fetch of a large number of topics is expected then fetch batching can be used. This allows fetches to be performed in batches with a specified time interval in between, which allows the client to process the messages and mitigate the possibility of the client queue filling.

The facility is enabled by configuring a fetch policy for the connector that the clients connect to. The policy can be configured in the `etc/Connectors.xml` configuration file or using the `FetchPolicyConfig` configuration object obtained from the connector. The policy comprises two settings, the batch size and the delay.

The batch size specifies the number of single topic fetch requests to be performed in a batch before waiting for the delay period to execute the next batch. If the batch size is specified as 0, no batching occurs.

Fetch handlers

You can assign a fetch handler to a topic. Such a handler is called whenever a client attempts to fetch the topic.

The handler is called after any other authorization processes and can be used to provide an additional layer of authorization.

You can delegate the fetch to some asynchronous process and defer replying until a response is received (see below).

If a fetch handler is in use for a topic, it is always used in preference to obtaining the topic data state or invoking the publisher to obtain the state. For this reason it is not normally used for topics that do have topic data.

The following example shows a fetch handler being used to prevent mobile clients from fetching the content a topic.

Note:

The handler has been applied to a topic that has topicData and so the actual state can be obtained from the TopicData:

```
TopicDefinition topicDef =
    new TopicDefinition(
        TopicDataType.SINGLE_VALUE,
        MetadataFactory.newFieldMetadata(MDataType.STRING));

topicDef.setProperty(
    TopicProperty.FETCH_HANDLER,

    new TopicFetchHandler() {

        @Override
        public TopicMessage fetchForClient(
        TopicClient client,
```

```
                Topic topic,
                List<String> headers) {

                    ConnectionType type =
  client.getConnectionType();
                    if (type.isCategory(ConnectionCategory.MOBILE))
  {
                        LOG.warn("Mobile clients not permitted");
                        return null;
                    }
                    try {
                        return topic.getData().getLoadMessage();
                    }
                    catch (Exception ex) {
                        ex.printStackTrace();
                        return null;
                    }
                }

            });

        addTopic("NonMobile",topicDef);
```

Asynchronous fetch using fetch handler

Another use of fetch handlers is to delegate the fetch to some asynchronous process and to return the reply later.

The following example shows the use of a fetch handler to delegate the fetch to an event publisher and send the reply when the publisher gets a response.

```
    @Override
    protected void initialLoad() throws APIException {

        TopicDefinition topicDef =
            new TopicDefinition(
                TopicDataType.SINGLE_VALUE,
                MetadataFactory.newFieldMetadata(MDataType.STRING));

        topicDef.setProperty(
            TopicProperty.FETCH_HANDLER,
            new TopicFetchHandler() {
                @Override
                public TopicMessage fetchForClient(
                TopicClient client,
                Topic topic,
                List<String> headers) {
                    try {
                        TopicMessage message =

 theEventPublisher.createDeltaMessage("FetchTopic");
                        message.setHeaders(headers);
                        message.putFields(
                            client.getClientID(),
                            topic.getName());
                        theEventPublisher.send(message);
                    }
                    catch (Exception ex) {
                        ex.printStackTrace();
                    }
                    return null;
                }
            });

        addTopic("Delegate",topicDef);
    }
```

```
    @Override
    protected void messageFromEventPublisher(
    EventConnection eventConnection,
    TopicMessage message) {
        if (message.getHeader(0).equals("FetchReply")) {
            try {
                Client client =
 Publishers.getClient(message.getHeader(1));
                if (client!=null) {
                    List<String> correlationId = new ArrayList<String>();
                    correlationId.add(message.getHeader(2));
                    client.sendFetchReply(message,correlationId);
                }
            }
            catch (Exception ex) {
                LOG.error("Error handling fetch reply",ex);
            }
        }
    }
```

The example shows the fetch handler sending a request to an event publisher containing the client ID and the topic to fetch. It then returns null so that no reply happens at this point.

Topic sets

Topic sets are a mechanism in the Java API for specifying a set of topic names and/or topic selectors.

Topic sets are provided by the issued `topicSet` class.

TopicSet extends the Java LinkedHashSet class and provides a set which maintains the order that elements were added to it. This can be important where the order that topic selectors are applied is important.

TopicSet also provides a validation mechanism. Its validate method can check that all entries are valid topic names and/or selectors.

Most API interfaces to which topics can be specified also have equivalent methods that will take topic sets.

Topic attachments

In the Java API, every topic can have an attachment. An attachment is an object which the user associates with the topic.

An attachment can be attached to a topic using the `attach` method and retrieved using the `attachment` method. To remove an attachment `attach(null)`.

If a topic attachment is relied upon for the topic state, attached it when the topic is created using the addTopic variant that takes a `TopicDefinition`.

Topic aliasing can be turned on or off for a publisher using a `etc/Publishers.xml` entry. For an event publisher it can be turned on or off using a `setTopicAliasing` method.

Though the data for a topic can be attached to a topic in this way the preferred method is to use topic data.

Chapter 7

Message conflation

In this section:

- Advantages of message conflation
- Types of message conflation
- How does conflation work?
- Configuring conflation
- Conflation counts

Conflation of messages is the facility to treat two messages as being essentially the same and avoiding sending duplicate information to clients. This involves removing an existing message from the outbound client queue and replacing it with a newer, equivalent message either at the position of the old message or at the end of the client queue.

For example, if a client application is being updated with the price of a product, several price changes can occur in rapid succession and be queued for the client. However, the client requires only the latest price, not any historic ones. Conflation is the ability to remove the superfluous messages before they are sent to the client, which reduces data transmission and client processing.

Diffusion also supports structural conflation, which is where the content of the existing queued message is merged with the content of the new message to produce a new merged message. This enables all of the data from the two messages to be sent to the client but as a single message. Structural conflation is also known as merging conflation.

Normally, a message is considered to be equivalent to another message if the messages belong to the same topic. However, messages of the same topic can only conflate messages according to some other criteria based on the message content.

Conflation policies configure how conflation takes place on topics. The policies can define the following behaviors:

- How messages are matched
- Whether replacement is done in place or by appending
- How to merge the two messages

Conflation is an optional feature that can be applied to all clients, clients connecting through a specific connector, or can be applied programmatically on a client-by-client basis.

Advantages of message conflation

There are many scenarios in real-time data distribution where data being communicated need not only be current, but must always be consistent. Structural conflation maximizes for concurrency whilst ensuring consistent delivery.

Concurrency

Both simple and structural conflation maximize for concurrency through avoiding distributing soon to be stale data. The higher the rate of change, the higher the value extracted. On other words, where clients or servers are running near to saturation based on available connectivity Diffusion can automatically adapt to this load by minimizing the data distributed.

In addition to the load-adaptive nature of conflation the effect is fair for clients connected to the same topic. The frequency of distributed changes is evenly amortized across clients.

Consistency

Structural conflation synthesizes complex event processing techniques in a highly efficient (lock-free wait-free concurrency) form inside the server. The specific knowledge of data structures and the semantic concerns for distributing data for a given topic in a given system allows consistent views of the data to be delivered in a way that is not possible with messaging technologies that treat messages as opaque.

Types of message conflation

The types of message conflation are simple, where a new message replaces an older message, and structural conflation, where the data from two messages is combined in accordance with a user-defined operation to create a new message. Both types of message conflation can either append the new message to the end of the queue or replace an existing message on the queue.

No conflation

Figure 17: Message flow without conflation enabled

With no conflation, a stream of messages to a client is delivered to the client in the order that they are published or sent. In example shown in the diagram, two messages for topic A, one message for topic B, and two messages for topic C are ready to send to the client.

This is a scenario common to all messaging platforms.

Simple conflation

When a new message is published or sent to a topic, simple conflation removes any earlier messages for that topic from the queue. The new message can be added to the queue either in the position of the earlier message that was removed (replace) or at the end, preserving the original message order (append).

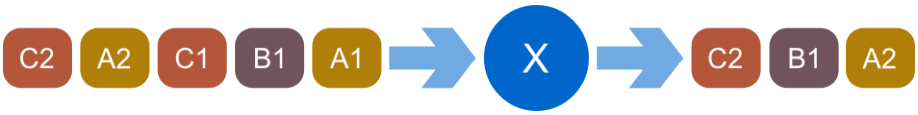


Figure 18: Message flow with simple replace conflation enabled

With simple replace conflation, only the most recent message for a topic is delivered to the client. In the example shown in the diagram, the first message published or sent to topic A is removed and the second message is added to the queue in the position that the first message occupied. The same occurs for the messages sent to topic C.

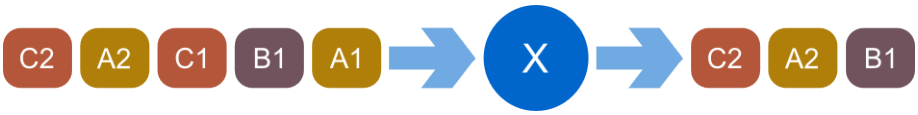


Figure 19: Message flow with simple append conflation enabled

With simple append conflation, only the most recent message for a topic is delivered to the client. Messages are delivered in time order. In the example shown in the diagram, the first message published or sent to topic A is removed and the second message is added to the end of the queue. The same occurs for the messages sent to topic C.

> **Note:** Use this option with care because there is the possibility that messages for a topic can continue to be added to the end of the queue and a value not be delivered for the topic.

In both the append and the replace example, although five messages were ready to send, only three messages were sent. This saves bandwidth and ensures clients receive current data only.

Structural conflation

Structural conflation allows a user-defined operation to be plugged into Diffusion so that rather than refreshing stale data with fresh data, a computation can be performed to merge, aggregate, reverse or combine the effects of multiple changes into a single consistent and current notification to the client.

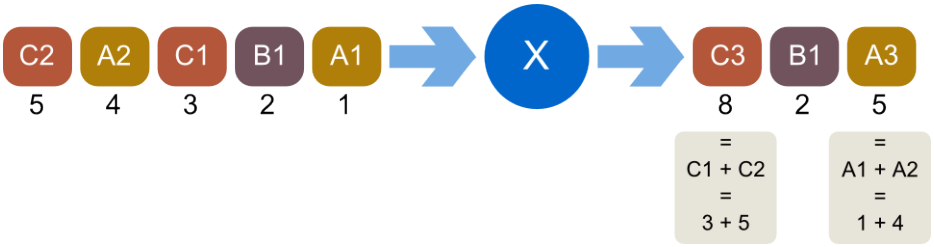


Figure 20: Message flow with merge and replace conflation enabled

In the example shown in the diagram, the operation is the summation of numeric data. The user provides the merge algorithm, that is the summing of the values of two successive messages, and Diffusion sends a single resulting message rather than the individual messages that were combined. The messages A3 and C3 are new messages generated from the merging process.

This is suitable for any scenario where the result is required but individual components that combine to form the result are not required.

The preceding example shows merge and replace. You can merge and append in a similar way as described for simple conflation above.

Various options are available to the user-written merger so that instead of returning a merged message it can indicate that either of the input messages be queued or that the no conflation option be chosen.

Selection of messages for conflation

The preceding examples assume that when a new message is queued for a client it replaces or merges with the last message queued for the message topic. This is the default behavior. When operating conflation in this way there is only ever one message per conflated topic awaiting delivery to the client.

You can specify a user-defined matcher that is used to determine the message that is to be replaced or merged with. This can be used to inspect the content of the messages queued for a topic to select which to conflate. When operating conflation in this way it is likely there can be more than one message per conflated topic awaiting delivery to the client.

Considerations when using conflation

- Conflation favors currency and reduces (but can not entirely eliminate) the delivery of stale data.
- When using conflation that alters message order it is assumed that there are no relationships or dependencies across the topics concerned. That is to say a topic is not temporally or causally related to any other topic.
- Consistency is tunable and a function of merge behavior. So for primitive data types a merge is fully consistent (for example, merge summation) but for complex data types (for example, a order book) this might be more complicated.
- Combining conflation with throttling maximally reduces bandwidth whilst enabling current and consistent data delivery to clients. Configuration of throttling along with conflation must be done with care.
- Do not use conflation of any kind when individual messages carry forensic storage or audit trail requirements.

How does conflation work?

Conflation is implemented as a lock-free, wait-free algorithm and can be scaled to meet demands of large numbers of concurrently subscribed clients. Slow clients, such as those on embedded devices or available over unreliable, low bandwidth connections receive less frequent updates than local area clients, or low latency high bandwidth clients.

When conflation is enabled for a client, every time a new message is enqueued for the client it is be considered for conflation.

The first step is to check if there is a conflation policy that has been mapped to the topic that the new message is for. If there is no policy, the message is added to the end of the queue as normal.

If there is a policy for the new message, the matching criteria of the policy is used to scan the queue (from back to front) for an existing message that matches the one being queued. If no match is found, the new message is queued at the end of the queue as normal.

> **Note:** Fast caches and lookup mechanisms are used to find policies and messages in the queue. The whole client queue is not scanned when looking for a match, only messages for the same topic. If the default matching (by topic) is used, there is not even any need for a comparison with the existing messages. This means that the conflation mechanism is as efficient as possible.

Having found a matching message in the queue, the conflation policy indicates whether the message to be queued is the new message or a message produced by merging the content of the current and new messages. The policy also indicates whether the message to queue (whether that be the new message or a merge result) replaces the existing message or whether the existing message is removed and the new message added to the end of the queue.

Conflation occurs on a client by client basis in the multiplexer thread. Results of merge actions are cached to ensure that merges of identical message pairs are not repeated for different clients.

Consider the following important points about conflation:

- Topic load messages are not conflated – only delta messages.

- Only Normal priority messages are conflated, not High or Low priority messages.
- Fragmented messages are not conflated.
- Messages requiring acknowledgment are not conflated.

Configuring conflation

Conflation is configured by defining one or more conflation policies which describe how the conflation is to be done and mapping topics to those policies.

Conflation policies can be configured in the `conflation` section of the `etc/Server.xml` properties file or can be configured programmatically using `ConflationConfig`.

Conflation policies

What is defined in a conflation policy?

One or more conflation policies can be configured, each defining different conflation mechanisms.

Conflation policies can be configured in the `conflation` section of `etc/Server.xml` using `conflation-policy` elements.

Conflation policies can also be configured programmatically using the various `addPolicy` methods on `ConflationConfig`. Such policies can also be added dynamically after the Diffusion server has started.

Conflation policies comprise the following:

Table 59: Conflation policy elements

Property	Description
name	A unique name by which the policy is referred to.
mode	Indicating whether the new (or merged) message is to replace the current message in place or whether the current message is to be removed and the new one appended to the end of the queue.
matcher	A Java class which matches two messages and is used to locate an existing queued message as a candidate for conflation. If no matcher is specified then default matching finds a message that is of the same topic.
merger	A Java class which performs the merge of two messages of the same topic to produce a new message containing the data from both messages (or any resulting data required). If no merger is specified, no merging takes place and the current message is removed from the queue and the new message either replaces it or is appended to the queue depending upon the mode. The merger can also indicate that either the current or new message is to be used or even that no conflation takes place in this instance.

Having defined one or more conflation policies, you can map topics to them. This is done by specifying a topic name or a topic selection string (regex pattern) which maps to a particular conflation policy.

Conflation policies can be added or removed at runtime and the removal of a conflation policy automatically removes any mappings to it.

Conflation policy mode

Modes of conflation

The conflation policy mode determines whether the new (or merged) message is to replace the existing message in the client queue or be appended to the end of the client queue.

Available modes are:

Table 60: Conflation policy modes

Mode	Definition
`REPLACE`	The new (or merged) message will replace the existing message at its current position in the client queue.
`APPEND`	The current message is removed from the client queue and the new (or merged) message is appended to the end of the queue.

If no mode is specified, `REPLACE` is assumed.

The mode is specified in the `mode` property of a `conflation-policy` section in `etc/Server.xml`.

When defining conflation policies programmatically the mode is specified when creating the policy.

Message matchers

How to use message matchers

A message matcher is used by a conflation policy when queuing a new message for a client that has conflation enabled for a topic that has a conflation policy defined for it. The message matcher is used to locate the last message queued for a client that is a candidate for conflation.

If no message matcher is explicitly defined for a conflation policy, a default matcher is used which locates a message of the same topic.

A message matcher can be supplied if the matching is to be somehow dependent upon the content of the messages.

To implement a message matcher, write a Java class that implements `com.pushtechnology.diffusion.api.conflation.MessageMatcher`. This has a single method called `matches` to which is passed the current message in the queue being tested and the new message to be queued.

> **Note:** The existing message is always of the same topic as the new message so you do not have to check that is the case.

An example of a `MessageMatcher` implementation is shown below:

```
public class ExampleMessageMatcher implements messageMatcher {
    @Override
    public boolean matches(TopicMessage currentMessage,TopicMessage
 newMessage) {
        return
 currentMessage.getHeader(0).equals(newMessage.getHeader(0));
    }
}
```

`MessageMatcher` implementations must be threadsafe and stateless. The same `MessageMatcher` instance can be supplied to more than one different conflation policy if so required.

Message mergers

How to use message mergers

A message merger can be specified on a conflation policy if the action of the policy is to merge the content of an existing queued message with the new message being queued. This technique can be used when message data comprises more than one data item and it is desirable to reduce the number of messages sent to the client whilst preserving the data from all messages.

If no message merger is specified for a conflation policy, the policy replaces the current message with the new.

To implement a message merger, write a Java class that implements `com.pushtechnology.diffusion.api.conflation.MessageMerger`. This has a single method called `merge` to which is passed the current message in the queue and the new message to be queued.

> **Note:** The existing message is always of the same topic as the new message so you do not have to check that is the case.

The action of conflation will depend upon the message that is returned from the merger method, as follows:

Table 61: Action depending upon merge result

Returned message	Action
A new message	It is assumed that the new message represents a merging of the data of the two messages input and so the returned message either replaces the current message in the queue or the current message is removed and the returned message added to the end of the queue, depending upon the policy mode.
The current message	The current message is retained at its current queue position and the new message is not queued.
The new message	The new message either replaces the current message in the queue or the current message is removed and the new message appended to the end of the queue depending upon the policy mode. This is effectively the same as the result that occurs if there is no merger.
Null	No conflation will occur. The current message remains where it is in the queue and the new message is appended to the end of the queue,

An example of a messageMerger implementation is shown below:

```
@Override
public TopicMessage merge(TopicMessage currentMessage,TopicMessage
 newMessage) throws APIException {
    TopicMessage result = topic.createDeltaMessage();
    Map<String,String> values = new LinkedHashMap<String,String>();
    while (currentMessage.hasRemaining()) {

 values.put(currentMessage.nextField(),currentMessage.nextField());
    }
    while (newMessage.hasRemaining()) {
        values.put(newMessage.nextField(),newMessage.nextField());
    }
    for (Entry<String,String> entry:values.entrySet()) {
        result.putFields(entry.getKey(),entry.getValue());
    }
    return result;
}
```

The above example merges two sets of key value pairs from the two messages with the new message values overwriting any duplicates in the current message.

`MessageMerger` implementations must be threadsafe and stateless. The same `MessageMerger` instance can be supplied to more than one different conflation policy if so required.

Default conflation policy

Using a default conflation policy

You can specify a default conflation policy that is used for any topics that do not have explicit policy mappings.

Use a default conflation policy only if you want to apply conflation to all topics when conflation is enabled for a client.

This can be specified using the `default-conflation-policy` property in the `conflation` section of `etc/Server.xml`. Alternatively it can be set programmatically at any time using the `setDefaultPolicy` method on `ConflationConfig`.

If no default policy is set, conflation will not occur for topics that have no explicit mappings even when conflation is enabled.

Mapping topics to policies

Mapping topics to conflation policies

Having defined one or more conflation policies, you can map topics to the conflation policies that are to be used for them.

Either a full topic name or a topic selector pattern can be used when mapping to a conflation policy.

As the use of topic selectors makes it possible for more than one mapping to potentially apply to the same topic, the last mapping defined that matches a specific topic is the one that is used for conflating messages of that topic.

A default conflation policy can be specified which is selected to map to if no other mapping matches a topic.

Messages for topics that have no mappings (when there is no default policy) are not conflated, even if conflation is enabled for a client.

Conflation mappings can be defined using `topic-conflation` elements within the `conflation` section of the `etc/Server.xml` property file.

Conflation mappings can also be set programmatically using the `setTopicPolicy` method of `ConflationConfig`. Mappings can be set at any time during the running of a server. Mappings can also be removed at any time using `unsetTopicPolicy`.

> **Note:** Removing a conflation policy at runtime will automatically remove any mappings to it.

Enabling conflation

This describes how to turn conflation on and off

Having configured the policies that define how conflation is to be performed on a per-topic basis, you can turn on conflation either for all clients connecting though a specific connector or for individual clients.

Enabling conflation means that conflation will occur for any messages queued for clients that have conflation policies set for one or more of their topics.

Enabling conflation using XML Configuration

Conflation can be specified for a queue-definition in the `etc/Server.xml` configuration file by setting the conflates property to true. This queue definition can then be used wherever required, for example by connectors that have conflation enabled for all clients.

Enabling conflation for a connector programmatically

When programmatically starting a Diffusion server then conflation can be specified in a similar way to within XML by setting the conflates property on a queue definition (`QueueConfig`).

Enabling conflation for a specific client

Conflation can be turned on (or off) programmatically at any time for a connected client. For example, you can choose to start conflating a client's message queue as a result of the queue's upper threshold having been exceeded. To turn conflation on for a client use the `setConflation` method on the `Client` interface.

Conflation counts

Determining whether conflation has occurred

To determine whether conflation has occurred for a client you can call the `Client.getConflationCount` method which returns the number of conflations that have occurred for the client.

This also has the facility to reset the count when called.

Chapter 8

Messages

In this section:

- Introduction
- Message types
- Creating messages
- Populating messages
- User headers
- Reading messages
- Records
- Byte encoding of a message
- Character encoding
- Message priority
- Acknowledged messages
- Fragmented messages
- Message filters
- Metadata

Publishers publish messages on topics which clients subscribe to so they can receive those messages. Clients can also send messages to publishers.

Messages are sent from event publishers and routed to the publishers that receive them. Messages are part of the basic Diffusion protocol and have the same underlying format no matter what platform or API is being used.

Introduction

Within user written applications messages must be created and populated with data for publishing or sending. The recipients of those messages can read them.

Special features of messages include the ability to change the delivery priority or request that they be acknowledged by the recipient.

Within publishers there are special facilities for comparing messages.

What is a message?

The make-up of a message

In Diffusion terms a message is a series of bytes with a variable length header. Its format is as follows;

Table 62: Message format

Message length	The length of the message including headers. This is a 4, 2 or 1 byte (space used is configurable) signed twos-complement integer.
Message type	A single byte identifies the type of message.
Byte encoding	A single byte indicates whether byte encoding is applied to the message.
Fixed headers	Optional fixed headers depending upon message type.
User headers	User headers which can be added to a message before adding data.
Data	Optional data.

Headers

The first 6 (4 or 3) bytes is a fixed overhead for all messages.

Whether there are any fixed headers depends upon the message type (see Protocol Specification for full details). User headers can be added as required.

Headers are variable length and are separated by field delimiters (byte value x02) and headers as a whole are terminated by a record delimiter (byte value x01). These delimiters are special values used within Diffusion Messages.

Headers are UTF-8 encoded character strings. Use of UTF-8 means that the characters used in header values (such as topic names) are not restricted to the ASCII set.

Data

The data part is a variable number of bytes. Diffusion has no interest in the content which is entirely user specific. It can be character data, encoded into bytes in a way that the user requires, or it can be any form of binary data. There are no restrictions on data content except for browser clients using WebSocket, XHR or iFrame in which messages are UTF-8 encoded.

The APIs provide facilities for writing data to messages and reading data from messages.

Message record and fields

Diffusion has the concept of records and fields within a message

Even though the data within a message is user specific, Diffusion has the concept of records and fields within a message which can simplify message handling, particularly within publishers.

Message capacity

Specifying the capacity of a message

Within the Diffusion server (and some APIs such as Java), Messages have internal message buffers. A message buffer is a fixed length area of memory assigned to make the handling of messages more efficient.

Whenever a message is created in Java, specify a message capacity (a default is used if the capacity is not specified). The capacity specified covers only the data and user headers (other overheads are be automatically calculated). To avoid unnecessary memory usage, specify the capacity as small as possible. If more data is written to a message than was specified to its capacity, the message is automatically extended but as there is an overhead to such extensions for maximum performance ensure the capacity is sufficient.

It is important to note that the message size and the message capacity are different and only the number of bytes indicated by the size are ever sent over connections even though each message consumes its capacity (plus an allowance for headers) in memory whilst it exists within the server (or Java API) environment.

In-bound messages are assigned buffers which exactly match the message size in their headers.

When a message is encoded, a second internal buffer is also assigned to the message. An inbound encoded message has an encoded buffer only until it is actually read at which point a decoded buffer is assigned.

Maximum message size

The concept of maximum message size exists within the server environment and within client side APIs

This is a system wide setting which specifies the largest size that any message can be (including headers). This value is also used in certain areas for tuning.

Within the server the maximum message size is specified in `etc/Server.xml`

In client side Java code the maximum message size is as specified using `ConfigManager.getConfig().setMaximumMessageSize()`. Call this method before a connection is made.

Message types

The different types of message specified by the Diffusion protocol

The Diffusion protocol specifies a number of different message types. Unless you are writing an application using the raw protocol, most of these message types are of no concern as they are largely hidden by the APIs. However, it can be of use to be able to identify the message types for diagnostic purposes.

This section discusses only those message types that are made visible through the APIs

Topic messages

The types of messages that are published on topics

Messages that are published on topics are one of two types – topic load and delta. Both of these types comprise a fixed header containing the topic name, optional user headers and data (which though typical is also optional). These two types are discussed below.

Topic load messages

A topic load message is what is (usually) sent to a client when it subscribes to a topic. The purpose of it is to provide the client with the current state of the topic data. The use of these is not mandated by Diffusion but provides a simple mechanism for dealing with what is typical functionality.

Because these messages represent the topic state they are typically created by a publisher but can also be sent to a publisher from an event publisher if required.

Such messages are typically created during topic loading. See the section on creating messages for more detail.

In the Java API this is a `TopicMessage` which returns `true` to the `isTopicLoad` method.

Delta messages

The word delta is conventionally used to describe change. A delta message is typically used by a publisher to publish changes in the state of a topic to all clients that are subscribed to that topic.

Delta messages can also be used by publishers to send messages to individual clients, and by clients to send messages to publishers.

A delta message does not contain the whole topic state when a change occurs. Ideally, only those items of data that have changed are sent. You must define some internal application protocol to handle this when designing a publisher. The use of hierarchical topics can simplify this if only a single data item is stored with each topic in the hierarchy.

How delta messages are created depends upon which API is in use. See the section on creating messages for more detail.

In the Java API this is a `TopicMessage` which returns true to the `isDelta` method.

Ping messages

Ping messages are used to test a connection exists, and its latency

Ping messages are special types of message used to ping the other end of a connection to establish that it is still there and also to measure latency. When one end of a connection sends a ping message it is the responsibility of the other end to reflect it back as soon as it is received.

A client can send pings to a server (server ping), a publisher can send a ping to a client (client ping), or the server can send regular pings to all clients (system ping).

> **Note:** This ping message handling is all done by the client libraries and APIs which provide mechanisms for sending such a message and being notified when its response is received.

Server ping

A server ping message is one sent from a client to a server which attaches the client queue size to it before reflecting it back to the client.

Client APIs provide a mechanism for sending such a ping message and receiving the reply.

For example, in the Java Classic API a server ping can be sent using the `ExternalClientConnection.ping` method and the is returned to any `ServerPingResponseListener` set using the `setPingResponseListener` method.

In the Unified API, you can use the Pings feature to ping the server from a client.

Client ping

A client ping message is one sent from a publisher to a client which the client reflects back so that the publisher can measure the round trip time. The client API automatically reflects such messages and the client application does not have to handle them. See client pings in the publisher section for details of how to initiate a client ping and receive its response

System ping

A system ping message is a message sent by the server to all connected clients at a regular interval. The interval can be configured by using the `system-ping-frequency` element in the `Server.xml` configuration file.

The client libraries automatically reflect such messages. You do not need to handle them within your client application.

The server uses these pings and the responses received to determine whether a client has become unresponsive. If a client does not respond to the system ping, the server closes its connection with a reason of CLIENT_UNRESPONSIVE.

Acknowledged messages

a publisher can set a message as requiring acknowledgment

Acknowledged messages are actually different message types in protocol terms because they carry an extra header holding the ack ID. However, from the point of view of the APIs this is not apparent.

For example, in the Java API, when a publisher sets a message as requiring acknowledgment its underlying message type is changed and an extra header is added, however, from the point of view of the API it is the same message. Upon receiving such messages the client APIs strip off the ack ID before converting the message type and passing the message onto the client application. The same occurs in the opposite direction (from client to server).

Control messages

messages which are unseen by users of the API

There are a number of other control messages within the protocol which are never seen by users of the API. For example the messages sent from a client to the server when it subscribes to topics.

For diagnostic purposes the various types of control message are all described in the protocol specification.

Creating messages

In protocol terms a message is a stream of bytes sent using the Diffusion Protocol

In API terms a message is an object which can be populated and sent or received and read. Different APIs approach message object creation in different ways. This section will use the Java API as an example. For other APIs consult the specific API documentation.

The mechanism for creating Topic messages are factory methods on various API classes. These allow you to create either topic load (using createLoadMessage) or delta (using createDeltaMessage) Messages.

In a publisher environment such methods exist on the publisher, the topic or there are static versions on the publishers class.

The topic that the message is for must always be specified when creating a topic message.

When a message is created an internal buffer is assigned to it and the initial size of the message data (its capacity) is specified on creation.

A topic can have default load and delta message capacities set so that it is not necessary to specify capacity on every message creation. If the capacity is not specified, configured defaults are used.

Where message capacities are the same or similar across a number of topics it might be more flexible to specify the capacity as a value obtained from a publisher integer property.

Topic load and delta messages can also be created with properties allowing them to be fragmented when sent from a publisher. Using createLoadMessage or createDeltaMessage, a fragment size can be specified. If the message exceeds this size, it can be split into smaller fragments which are sent individually and recombined by the client. Additionally, a delay between sending each fragment is given to allow large messages made of many fragments to be interleaved with other messages. A delay of zero indicates that the messages are sent as quickly as possible.

In APIs other than the publisher API there are various factory methods for creating messages as follows;

Both the ExternalClientConnection (client API) and PublisherServerConnection (publisher API) represent client connections to a Diffusion server and as such are both of type ServerConnection.

A ServerConnection has a convenience method called createDeltaMessage to create messages. It is normally best to supply a capacity when creating messages to avoid unnecessary automatic extension of message.

The EventPublisherConnection class has createLoadMessage and createDeltaMessage methods which operate in a similar way to the client API.

Populating messages

A Diffusion message comprises Diffusion headers, user headers and user data. When using an API, after creating a message, the message must be populated with user header and/or user data.

User headers are optional and there can be any number of them as required. They must be added to the message before any data is added. Headers are UTF-8 encoded character data.

Data is byte data and can be used for any purpose that you require. If the user wants to write character data, the data is typically UTF-8 encoded but any character encoding can be used if desired. Diffusion has no interest in the data content but the APIs do provide some utilities that make populating and reading message data much easier.

Diffusion provides the facility for compressing or encrypting message data for transmission. This occurs at a byte level (the bytes within the message have compression or encryption applied to them) and is referred to as the message's byte encoding. For more information, see Byte encoding of a message on page 206.

Each language API can address the populating and encoding of messages differently. The remainder of this section discusses populating message using the Java API (that is, within a publisher, or when using other external Java APIs).

Setting user headers

User headers can be set on a message using the setHeaders method which is a varargs method. Any number of String headers can be set in one call. This can be called only once and must be called before any data is added to the message. There is also a version of this method that takes a List parameter.

```
TopicMessage message = topic.createDeltaMessage(20);
message.setHeaders("Header1","Header2");
```

Setting data

In the Java API data is put into messages using the various `put` methods. When a message is newly created it has a pointer set just after the message headers and a put writes data at the pointer position and update the pointer position.

It is important to note that once a message has been published (or sent) it can no longer be updated.

Setting String content

If you want to write character data to a message, the `put(String...)` method can be used. As this is a varargs method you can supply any number of Strings and they are concatenated in the message. So, for example

```
TopicMessage message = topic.createDeltaMessage(6);
message.put("AA","BB");
```

```
message.put("CC");
```

Each call to `put` updates a pointer in the message so that the next `put` (or any other message populating method) starts from where the last one ended. So the above example results in a message with data content of:

AABBCC

> **Note:** What is actually written to the message is the byte representation of the Strings encoded using the default character set for the message (see character encoding). The number of bytes used in the message can be more than the number of characters put.

Setting byte content

For non-character data you can write bytes directly to the message using either the put(byte) or put(byte[]) methods.

Using records and fields

To simplify the handling of character data, Diffusion has the concept of records and fields. See the section about Records for more information.

User headers

User headers can be set on a message using the setHeaders method which is a varargs method. Any number of String headers can be set in one call. This can be called only once and must be called before any data is added to the message. There is also a version of this method that takes a List parameter.

```
TopicMessage message = topic.createDeltaMessage(20);
message.setHeaders("Header1","Header2");
```

User headers can be set on a message using the setHeaders method which is a varargs method. Any number of String headers can be set in one call. This can be called only once and must be called before any data is added to the message. There is also a version of this method that takes a List parameter.

```
TopicMessage message = topic.createDeltaMessage(20);
message.setHeaders("Header1","Header2");
```

Setting String content

User headers can be set on a message using the setHeaders method which is a varargs method. Any number of String headers can be set in one call. This can be called only once and must be called before any data is added to the message. There is also a version of this method that takes a List parameter.

```
TopicMessage message = topic.createDeltaMessage(20);
message.setHeaders("Header1","Header2");
```

Setting byte content

For non-character data you can write bytes directly to the message using either the put(byte) or put(byte[]) methods.

Using records and fields

To simplify the handling of character data, Diffusion has the concept of records and fields. For more information, see Records on page 203.

User headers

User headers are a feature of Diffusion messages that allows String header values to be set in addition to and separately from the message's data payload.

User headers offer a convenient mechanism for adding extra control information to a message.

User headers are typically set after creating a message and before populating with other data. It is permissible for a message to have only user headers and no data.

In terms of the message content user headers are held after(and in the same format as) any fixed headers.

User headers are String format. The user headers are UTF-8 encoded and support the full Unicode character set.

User header values can be set in a message using the APIs. Each API can differ in the way that this is done.

The use of user headers within the Java API is outlined below.

Adding user header data

User header values must be set after creating a message and before populating it with any data as follows

```
TopicMessage message = topic.createDeltaMessage(10);
message.setHeaders("H1","H2");
message.put("data");
```

Retrieving user headers

User header values must be set after creating a message and before populating it with any data as follows

```
TopicMessage message = topic.createDeltaMessage(10);
 message.setHeaders("H1","H2");
 message.put("data");
```

Retrieving user header data by index number

You can access header values individually

```
String h2 = message.getHeader(1);
System.out.println(h2);
```

Note: Header index numbers start from 0 so the second user header value is obtained using index number 1. If a header with the requested index does not exist, null is returned.

Reading messages

Whether within a client or a publisher the various APIs provide mechanisms for easily reading the content of messages. The way in which this is done varies across different APIs.

Diffusion Messages are typically created and populated by publishers and published to Clients that read the message content. Messages can also be created by clients (or other APIs such as the event publisher) and sent to the publisher.

This section outlines how message content can be read in the Java API, for example within Publishers or when using the Java client API.

Reading messages in the Java API

The data content of a message (as opposed to the User headers) can be read in a number of different ways. The most suitable to use depends upon the nature of the data and how it has been structured within the message.

Encoding considerations

> **Note:** Though messages might have been byte encoded for transport, before they are read they are automatically decoded. Because of this, you do not have to consider the fact that the underlying message might be encoded.

When reading character data (Strings) from messages the character encoding must be considered. Character reads automatically convert message bytes to characters using the message's defined character set. As the default is normally UTF-8, no special handling of character sets is usually required.

Message pointers

Message data reads can be absolute or relative. With an absolute read the data is read positionally from the message. With a relative read, data is read relative to a pointer maintained within the message. When an incoming message is presented to an API by Diffusion, its pointer is set at the start of the message data. Each relative read will move the pointer to be positioned after the data read. Absolute reads have no effect on the pointer. The pointer can be reset to the start of the data at any point using the rewind method.

Reading whole message content

The whole data content of a message can be read as a String or as an array of bytes.

Content as a String

To read as a String

```
String data = message.asString();
```

The above call converts all of message data bytes to a String using the message's default character set.

It is important to note that the length of the returned String might not be the same as the length returned by the length method of the message. This is because only standard (for example, ASCII) characters can be encoded as a single byte, so if non-standard characters are in use the returned String length might be shorter than the message length.

Content as bytes

To read a message as bytes

```
byte[] bytes = message.asBytes();
```

In this case the byte array returned is exactly the same length as the value returned by the message length method.

Both of the above examples are absolute reads and do not affect the message pointer.

Traversing message content

You can traverse the content of a message in a relative manner using the `next` set of methods on the message. Each time one of these methods is used the message pointer is updated so that it points to after the data that has been read.

Reading bytes

Use the `nextByte` method to read a message byte-by-byte. To read blocks of bytes use the nextBytes method. The `nextBytes` method is useful for fixed format messages. In both cases an exact number of bytes are read starting from the current message pointer.

The `remaining` method can be used to determine how many bytes of data there are left to read in the message at any time. The `hasRemaining` method indicates whether any bytes remain to be read.

It is important to determine whether there are sufficient bytes to read before invoking `nextByte` or `nextBytes` as an exception will occur if an attempt is made to read off the end of the message.

Reading one byte

The following example shows how to read message data one byte at a time. In this example each byte value is displayed as a decimal value.

```
byte b;
while (message.hasRemaining()) {
 b=message.nextByte();
 System.out.println(Byte.toString(b));
}
```

Reading array of bytes

The following example shows how to read a message in 4 byte segments and display the hexadecimal values of each 4 bytes using the Diffusion `Utils` class

```
byte[] bytes = new byte[4];
while (message.remaining()>3) {
 message.nextBytes(bytes);
 System.out.println(Utils.bytesToHex(bytes,0,4));
}
```

In both of the above examples it is assumed that the message pointer is positioned at the start of the data. This is where the pointer is positioned when the message is sent to the client by Diffusion or if `message.rewind()` had been called.

Reading records and fields

If a message has been populated using the Diffusion field and/or record delimiters, the nextRecord and nextField methods make the reading of these records and fields simple. For more information, see Records on page 203.

Concurrency issues

When working in a multi-threaded application it is important to note that the relative methods of a message are not threadsafe. If more than one thread accesses a message at the same time, they might both be updating pointers at the same time. One way of overcoming this might be to take a copy of the message (using TopicMessage.duplicate) but this is inefficient and uses additional memory. A better mechanism is to use message readers.

Using a message reader

To read a message safely within a thread when it might also be accessed in another thread a MessageReader can be used. A MessageReader is an object that maintains its own read-only view of a message without affecting that message's pointers.

To obtain a MessageReader use the `getReader` method on a message. The reader has all of normal message read methods so you can access the reader in the same way as if it were the real message.

Note: A reader maintains a reference to the message it was obtained from, which prevents the garbage collection of that message whilst the reader is still in use.

Records

Even though the data within a message is user specific, Diffusion has the concept of records and fields within a message which can simplify message handling, particularly within publishers.

Records and fields relate to the handling of character based messages as they depend upon the use of separator bytes which are outside the character representation range. They cannot be used with binary data.

The separators used are

Table 63: Separator bytes

Field delimiter	Single byte value of hexadecimal 02
Record delimiter	Single byte value of hexadecimal 01

A notation of `<FD>` for field delimiter and `<RD>` for record delimiter can be seen throughout this manual

A field is zero or more characters terminated by `<FD>`, `<RD>` or the end of the message

A record is zero or more characters terminated by `<RD>` or the end of the message

In either case, if a message ends with an `<FD>` or an `<RD>` it implies an empty field or record at the end of the message

This representation is chosen to simplify the splitting up of character data.

Note: Because of the format, you cannot distinguish between an empty message and one with a single empty field or a single empty record and this must be borne in mind when designing message formats.

The Java API provides record and field handling capabilities that greatly simplify the handling of character based messages. Also, when using the Java API, records can have metadata associated with them, allowing fields to be populated and read by name.

The delimiters are handled as separators and not terminators so that they can easily be used to split up a String that contains them (see 'Reading messages').

Note: There is no difference between the internal representation of an empty message and a message with a single empty record and the use of such delimiters must never assume that this distinction can be made.

Note: Records are handled as separate objects and are not a map onto the message itself. This means that they can be manipulated between puts which can simplify the handling of data. Changes to a record after putting it into a message are not reflected in the message.

Unlike messages, Records have methods allowing Fields to be inserted, removed and replaced.

Populating messages

The Java API handles Records and Fields as follows.

Adding fields to a message

Fields provide a further level of Structure to a message

Fields are Strings which are written to the message separated by field delimiter bytes. Using these field delimiters separates the character data into variable length String fields within the message.

So, contrast the example of setting String content with the following example of using fields

Adding fields to a TopicMessage

```
TopicMessage message = topic.createDeltaMessage(10);
message.putFields("AA","BB");
message.putFields("CC");
```

The resultant TopicMessage consists of 3 fields. The field delimiter that is used is defined by the constant Message.FIELD_DELIMITER. If we represent the field delimiter as `<FD>` (though it occupies only one byte) then what is actually written to the message in this case is: `AA<FD>BB<FD>CC`

In this case a total of 8 bytes are written to the message. There is no delimiter at the start of the message data nor at the end. Writing fields in this manner greatly simplifies reading the data from the message and avoids the space wastage of using message definitions with fixed length fields.

Adding records to a message

Records provide a further level of Structure to a message.

Records are best thought of as groups of Fields separated by a special record delimiter.

This provides for the ability to easily handle structured data as records.

For example, a message can contain a variable number of customer records, each containing 'Customer Number' and 'Customer Name'. You can write such a message as follows

Adding records to a TopicMessage

```
TopicMessage message = topic.createDeltaMessage(40);
 message.putRecords(
  new Record("123","John Smith"),
  new Record("124","Fred Bloggs"));
```

The record delimiter that is used is defined by the constant Message.RECORD_DELIMITER. If we represent the record delimiter as `<RD>` (though it occupies only one byte), what is actually written to the message in this case is: `123<FD>John Smith<RD>124<FD>Fred Bloggs`

As with Fields, note that there is no delimiter at the start or end of the Message

Adding a record without using a Record object

Records can be manipulated as a separate object or can be added easily to a message using the `putRecord(String...)` method.

Add multiple fields as a record

```
 message.putRecord("125","A.N. Other");
```

Building upon the previous example we now have a message containing `123<FD>John Smith<RD>124<FD>Fred Bloggs<RD>125<FD>A.N. Other`

Adding an empty record to a message

It is permissible to put empty records into a message.

Add multiple fields as a record

```
message.putRecords(new Record(),new Record());
```

We now have a message containing `23<FD>John Smith<RD>124<FD>Fred Bloggs<RD>125<FD>A.N. Other<RD><RD>`

Note: In this case the final `<RD>` implies a blank record after it.

Reading messages

If a message has been populated using the Diffusion field and/or record delimiters, the nextRecord and nextField methods make the reading of these records and fields simple

Getting fields from a message

If the message is made up only of fields, the fields can be processed in order using nextField

```
String field;
while (message.hasRemaining()) {
 field=message.nextField();
 System.out.println("Field="+field);
}
```

Note: The above examples is not a reliable way of processing if there is the possibility of empty fields at the end of messages

Getting records from a message

When records are in use, the records can be read one at a time using nextRecord

```
Record record;
while (message.hasRemaining()) {
 record=message.nextRecord();
 System.out.println("Record="+record);
}
```

Note: The above examples do not show a reliable way of processing if there is the possibility of empty records at the end of messages. For other examples, see Catering for empty fields and records.

Looping through records and fields

Records also provide a convenient way of accessing data positionally as the fields of each record read can be accessed by their index. The following example demonstrates this by displaying only the content of the third field of each record read

```
while (message.hasRemaining()) {
 record=message.nextRecord();
 if (record.size()>2) {
  System.out.println("Field 3="+record.getField(2));
 }
}
```

We now have a message containing `23<FD>John Smith<RD>124<FD>Fred Bloggs<RD>125<FD>A.N. Other<RD><RD>`

Note: In this case the final `<RD>` implies a blank record after it.

Catering for empty fields and records

The method `hasRemaining` determines whether the end of the message has been reached. Because this method indicates whether there are any bytes left to read in the message it works as long as there is not the possibility of there being an empty field or record at the end of the message.

An empty field is one that is of zero length and the last byte of a message can be a field delimiter implying the presence of an empty field after it. However, as the nextField method always positions after the last field read, hasRemaining does not detect the presence of an empty field at the end of a message.

An empty record is one that has no fields within it. A record delimiter at the end of a message implies an empty record following. As with fields, checking for end of processing with hasRemaining does not detect an empty record at the end of the message.

Both nextField and nextRecord return null when there is no more to process. If there is an empty field at the end of a message, nextField returns a zero length String (even though hasRemaining returns false). If there is an empty record at the end of a message, nextRecord returns an empty record (one with zero fields).

Traversing fields

Traversing the fields of a message when the last field might be empty.

```
while ((field=message.nextField())!=null) {
 System.out.println("Field="+field);
}
```

Byte encoding of a message

Diffusion messages comprise some header information followed by some byte data. By default this is how the messages are transmitted over a connection.

Diffusion provides a number of byte encodings that can be applied to the data part of a message when it is sent over a connection. You can use the Classic API to enable your user applications to specify the encoding on a per-message basis.

Only one byte encoding can be specified per message. If you specify an encoding for a message and specify another encoding, only the second encoding that you specified is applied.

The encodings are listed in the following table:

Table 64: Types of byte encoding

Compressed	The data is compressed using the ZLIB compression library. This compression is effective in compressing sections of text.
Encrypted	The data is encrypted using an algorithm based on XXTEA.
Base64	The data is encrypted in base-64 notation.

The Diffusion server handles the encoding and decoding (or compression and decompression) of messages. User applications are not required to handle encoding and decoding (or compression and decompression).

Because the server handles the encoding and decoding (or compression and decompression), it is transparent to the user application.

Note: Some APIs do not support all of these encodings. For more information, see the API documentation.

Specify the byte encoding

You can specify the default encoding for a message after it is created. The encoding is applied to the message by the Diffusion server only if the server determines that the connection the message is transmitted through can support the encoding.

The following examples demonstrate how to specify byte encodings for a message in the Classic API:

Java

```
// Create a message
```

```
TopicMessage message = topic.createDeltaMessage(10);

// Set encryption on the message
message.setEncoding(com.pushtechnology.diffusion.api.message.Encoding.ENCRYPT);

// Set compression on the message
message.setEncoding(com.pushtechnology.diffusion.api.message.Encoding.COMPRESS);

// Set base 64 on the message
message.setEncoding(com.pushtechnology.diffusion.api.message.Encoding.BASE64);

// Note: Encodings are exclusive. Only one encoding, the last specified,
 is applied to the message.
```

.NET

```
// Create a message
ITopicMessage message = MessageFactory.CreateDeltaMessage( "Topic" );

// Set encryption on the message
message.SetEncoding( (byte)PushTechnology.DiffusionCore.Enums.MessageEncodingType.CRY

// Set compression on the message
message.SetEncoding( (byte)PushTechnology.DiffusionCore.Enums.MessageEncodingType.COM

// Set base 64 on the message
message.SetEncoding( (byte)PushTechnology.DiffusionCore.Enums.MessageEncodingType.BAS

// Note: Encodings are exclusive. Only one encoding, the last specified,
 is applied to the message.
```

JavaScript

```
// Create a message
var topicMessage = new TopicMessage("Topic", "Message content");

// Set encryption on the message
message.setCrypted();

// Note: Compression and base 64 encodings are not available in the
 JavaScript API.
```

ActionScript

```
// Create a message
var topicMessage:TopicMessage = new TopicMessage("Topic", "Message
 content");

// Set encryption on the message
message.setEncoding(TopicMessage.ENCRYPTED_ENCODING);

// Set compression on the message
message.setEncoding(TopicMessage.COMPRESSED_ENCODING);

// Set base 64 on the message
message.setEncoding(TopicMessage.BASE64_ENCODING);

// Note: Encodings are exclusive. Only one encoding, the last specified,
 is applied to the message.
```

Silverlight

```
// Create a message
TopicMessage message = new TopicMessage( "Topic", "Message content" );
```

```
// Set encryption on the message
message.setEncoding(PushTechnology.Transports.MessageEncodingType.EncryptionRequested

// Set compression on the message
message.setEncoding(PushTechnology.Transports.MessageEncodingType.CompressionRequeste

// Set base 64 on the message
message.setEncoding(PushTechnology.Transports.MessageEncodingType.Base64Requested);

// Note: Encodings are exclusive. Only one encoding, the last specified,
 is applied to the message.
```

iOS

```
/* Create a message */
DFTopicMessage *topicMessage = [[DFTopicMessage alloc]
 initWithTopic:@"Topic"
 andString:@"Message content"];

/* Set encryption on the message */
topicMessage.encoding = DIFFUSION_MESSAGE_ENCODING_ENCRYPTED_ENCODING;

/* Set compression on the message */
topicMessage.encoding = DIFFUSION_MESSAGE_ENCODING_COMPRESSED_ENCODING;

/* Note: Encodings are exclusive. Only one encoding, the last specified,
 is applied to the message.
   Base 64 encoding is not supported for iOS.  */
```

Android

```
// Create a message
TopicMessage message = new TopicMessage("Topic", "Message content");

// Set compression on the message
message.setEncoding(com.pushtechnology.mobile.enums.EncodingValue.COMPRESSED);

// Note: Encrypted and base 64 encodings are not supported in the Android
 API.
```

Transport capabilities

When a client connects to the Diffusion server, the server establishes whether or not the connection has the capability to support the encodings. If the connection can support the encoding that is specified for a message, the Diffusion server applies that encoding to the message. If the connection does not support the encoding, the message is transmitted unencoded.

The following table lists the encodings supported by each transport:

Table 65: Encoding support transports

Transport	Compressed	Encrypted	Base64
DPT	YES	YES	YES
WebSocket	NO	NO	NO
HTTP Duplex	YES	YES	YES
HTTP Comet	YES	YES	YES
HTTP	NO	NO	NO
iFrame	NO	NO	NO

Character encoding

Diffusion messages are made up of some header information followed by the data payload. From the point of view of Diffusion those data are only raw bytes but the representation of characters (normal textual information) as bytes depends upon the character encoding used.

For simplicity it is recommended that the UTF-8 character encoding is used and this is the default for all Diffusion API character handling. Message headers are always UTF-8 encoded. The encoding of character data is dependent upon the API but the default is always UTF-8.

To use any other character sets, you can change the default used in the Diffusion server by editing in the `etc/Server.xml` file.

The way in which the character encoding used with messages is handled can vary across APIs. The handling within the Java API is outlined below.

When in a Diffusion server environment (within publishers) then the default character encoding that is used when any String data is written to or read from a message is as specified in the `etc/Server.xml`file.

In a Java client side environment the default character encoding can be set using `ConfigManager.getConfig().setCharset()`.

These default settings can be overridden at an individual message level using the `setCharset` method on the message.

Using `getCharset` returns the character encoding that the message uses to encode or decode characters.

> **Attention:** It does not return the character set that might have been used to encode a message that has been received over a connection as this information is not passed in the message protocol.

UTF-8 is the default character set used within Diffusion.

The effect of character encoding on message sizing

Character encoding can result in a discrepancy between the number of data characters written to a message and the actual number of bytes used to represent that data. This is because certain characters cannot be represented in a single byte. For example, UTF-8 can encode any Unicode character but only some of those (for example, ASCII characters) can be held as a single byte. Other characters consume between 2 and 4 bytes. Consider this when sizing for messages that might hold any character data outside the normal ASCII range.

Message priority

You can send or publish messages at different priorities to either send them to the front of the queue or specify that they can be delivered after other messages on the queue.

Under normal circumstances all messages published or sent to clients by publishers are put on the client's queue and the messages are delivered to the client in the order that they are queued. However, there might be times when a message needs to be sent to a client urgently and goes to the front of the queue or there might be messages that can be delivered only when nothing else is queuing. This is achieved by sending or publishing the message at a different priority.

Normal priority

If a priority is not explicitly specified, normal priority is assumed. This causes a message to be queued after any existing normal priority messages.

All high priority messages are delivered before normal priority messages are considered for delivery.

All normal priority messages in a client queue are delivered before low priority messages are considered for delivery.

Only normal priority messages are considered for conflation.

High priority

A message can be sent with high priority causing it to go to the front of the client queue.

A message sent with high priority is queued after any high priority messages already queued and is delivered before any normal or low priority messages.

A typical use of high priority messages might be when the client queue has built up because the client is not processing messages fast enough. This might have been notified to a publisher by means of the clientQueueThresholdReached client notification.

High priority example

The publisher can inform the client that this has happened by means of a high priority message as follows

```
public void clientQueueThresholdReached(
 Client client,
 boolean upper,
 int threshold) throws APIException {

 if (upper) {
  TopicMessage message = topic.createDeltaMessage(50);
  message.put("Client Queue Reached Threshold of "+threshold);
  client.send(message,MessagePriority.HIGH);
 }
}
```

Low priority

A message can be sent with low priority so that it does not interfere with the delivery of normal messages. Low priority messages are considered for delivery only when there are no high or normal priority messages queued.

Acknowledged messages

You can specify that messages sent by the publisher to a client are acknowledged by the client when they are received.

Under normal circumstances, when a message is sent to a Client by a Publisher it is queued for the client and delivered using the normal queue and send mechanisms employed by Diffusion. The publisher is not notified of delivery of the message to the client.

There is, however, the facility to force client acknowledgment of messages sent by a publisher by using the `setAckRequired` method on a message. This assigns an ack ID to the message and when it is published (or sent), all clients that receive the message must acknowledge it within a specified time period otherwise the publisher is notified of non-acknowledgment. When the publisher receives notification of non-acknowledgment it is supplied with the message ack ID and a list of the clients that did not acknowledge the message.

The same facility exists from client to server. A client can set ack required on a message it sends, in which case the server (or publisher) must respond within a given time otherwise the client application is notified of non-acknowledgment of the message

Use message acknowledgment sparingly (see considerations below).

Setting messages as requiring acknowledgment

A message is set as requiring acknowledgment using the `setAckRequired` method. This must be called before any user headers or data are added to the message

For example,

```
TopicMessage message = topic.createDeltaMessage(4);
message.setAckRequired();
message.put("Data");
publishMessage(message);
```

After setAckRequired has been called the message's ack ID can be obtained using the getAckId method.

Once a message requiring acknowledgment has been published or sent it cannot be reused (sent again). Each ack ID can be used only once. If it is required to set up such a message and send several times, the message must be copied using the duplicate method which assigns a new ack ID.

When a message sent from a publisher is set as requiring acknowledgment, it is automatically sent with high priority and is not conflated.

Setting acknowledgment timeout

The time period within which a message must be acknowledged can be configured for a publisher in `Publishers.xml`. A publisher can programmatically override the configured default timeout using the `setAckTimeout` method on the publisher.

At the client end the default ack timeout can be set for the client connection.

The timeout can also be explicitly specified for a message using the setAckTimeout method on the message itself.

Acknowledgment timeout and publisher notification of non-acknowledgment

When Diffusion receives a message from the publisher with the ack flag set, a timer starts and Diffusion queues the message. If the client does not respond with an ACK before the timer elapses, or the message has not been queued before the ACK timeout expires, a NAK message is generated which notifies the publisher. It becomes the publisher's responsibility to resend the message.

The message ack timeout clock runs from the time the message is sent to Diffusion by the message broker, as opposed to when it is physically put on the wire to a specific client. As a result, if a client is on the end of a slow network connection and as the message queue continues to build, messages nearing the head of that queue can be nearing their timeout. When such a queued message reaches its timeout, it is discarded and the publisher is informed by the generated NAK.

Client handling of acknowledgments

From the client APIs point of view, Message acknowledgments can be handled automatically or manually.

When handled automatically, whenever a message requiring acknowledgment is received from the server, an ACK message is automatically sent back to the server. This mode involves no special processing in the client application.

When handled manually, when a message requiring acknowledgment is received it is not automatically acknowledged and it is up to the client application to acknowledge it. The acknowledgment can occur after the client application has processed the message. There is a method (`isAckPending` in Java) on messages to identify those not acknowledged and a method on the connection (acknowledge in Java) to manually acknowledge the message.

Server handling of acknowledgments

Users of the publisher API can cause message acknowledgments to be handled automatically or manually.

When handled automatically, whenever a message requiring acknowledgment is received from a client, an ACK message is automatically sent back to the client. This mode involves no special processing in the publisher application.

When handled manually, when a message requiring acknowledgment is received it is not automatically acknowledged and it is up to the publisher to acknowledge it, possibly after processing it. The `isAckPending` method on a Message indicates whether it must be acknowledged. Such a message can be acknowledged within a publisher using the `Client.acknowledge` method which sends an ACK notification back to the client.

Whether a publisher operates in automatic or manual acknowledgment mode is determined by the `auto-ack` property in `etc/Publishers.xml` .

Publisher notification of Non-acknowledgment

A publisher receives notifications of non-acknowledgment of messages through its messageNotAcknowledged method.

For example

```
public void messageNotAcknowledged(
    TopicMessage message,
    List<TopicClient> clients) {
        LOG.warn("Message {} not acknowledged by: ",message);
        for (TopicClient client:clients) {
            LOG.warn("Client '{}'",client);
        }
    }
```

Client notification of Non-acknowledgment

A client that sends messages requiring acknowledgment to the server must declare a listener for non-acknowledgment notifications. In the Java API the connection has a setAckListener method for this purpose). Any attempt to send a message requiring acknowledgment when no listener has been declared fails.

Message acknowledgment considerations

Use message acknowledgment sparingly and with care, taking the following into consideration:

- A message set as requiring acknowledgment cannot be reused unless it is copied (using duplicate). Efficiencies of message reuse (for example, using the same topic load message for many subscriptions if the data has not changed) are lost.
- A message set as requiring acknowledgment by a publisher is always sent with high priority and the normal client queuing mechanisms are by-passed. Queuing of expedited messages is less efficient than normal queuing as such messages have to be inserted at the head of the queue rather than being added at the end of it.
- Acknowledged messages cannot be conflated.
- Acknowledged messages cannot be fragmented.
- The handling of acknowledged messages involves extra processing at both the client and at the server. This has performance implications.

Diagnostic considerations

Messages requiring acknowledgment are actually a different message type (see protocol) from normal topic messages so from a diagnostic point of view, bear in mind that the message type number is different and there is an additional (Ack ID) header.

Fragmented messages

Fragmentation enables a publisher to send a message in smaller chunks over a period of time, and for that message to be reconstituted by the client.

Although most Diffusion messages are small, it is sometimes necessary to send messages larger than the normal limits. You can use fragmentation to do this. Advantages of this technique are

- Message size is not constrained by the configured maximum message size
- Other messages can be interleaved, so that the sending of one very large message does not block subsequent messages.

Creating a fragmented message

Fragmented messages can be created by setting the fragmentation settings of a topic message. The message is fragmented only if it is larger than the fragment size.

Table 66: Creating a fragmented message

`TopicMessage message = createLoadMessage(size); message.setFragmentLifeCycle(new FragmentedMessageLifecycle(delay)); message.setFragmentSize(fragmentSize);`	Creates a new, empty initial load TopicMessage with a fragment lifecycle and a fragment size.
`TopicMessage message = createDeltaMessage(size); message.setFragmentLifeCycle(new FragmentedMessageLifecycle(delay)); message.setFragmentSize(fragmentSize);`	Creates a new, empty delta TopicMessage with a fragment lifecycle and a fragment size.

The size parameter is the maximum size of the message (excluding headers).

The lifecycle is an object which provides control over how the message fragments are queued over time:

Table 67: Fragmented message lifecycle

`FragmentedMessageLifecycle(int sendingDelay);`	Specify a delay (in milliseconds) between sending each fragment of a message.
`FragmentedMessageLifecycle(int sendingDelay, long maximumAge);`	Also set the maximum amount of time (in milliseconds) that is allowed to pass after the first message is queued before we consider the message to be too old to be of any use to the clients. After this time has elapsed, a cancellation message is sent to subscribed clients.
`FragmentedMessageLifecycle(int sendingDelay, long maximumAge, long priorityBumpTime);`	As above, but if the messages are still sending after priorityBumpTime milliseconds (which must be less than the maximum age), the priority of remaining fragments is increased.

The fragmentSize parameter is the size of the fragments the message is broken into.

Notes

- Ensure that you set the fragment size of the message, using `setFragmentSize()`, before you populate the message data. If you do not, an exception occurs when the message is larger than the maximum message size.
- A `MessageException` is thrown if you try to set the fragment size larger than the maximum message size.
- The fragment size can be set larger than the initial size of the message as a message can be resized if it exceeds the initial size.
- A size of -1 turns off fragmentation for the message, and a size smaller than 4 is illegal. Although very small fragment sizes are allowed, their use is discouraged. Typically, you request a fragment size as large as is realistically possible.
- If the sendingDelay is 0 (or less), there is no delay between sending each fragment and no other fragments or messages have the opportunity to be interleaved with this message.
- The FragmentedMessageLifecycle class has accessory methods for manipulating the above values to provide a different level of control, for example, setting the priority bump time while allowing a non-expiring maximum age for the fragments.

Message filters

Message filters are a feature within the publisher API which provide the ability to accept, reject, or replace a message being processed.

A message filter is a user-written class which must implement the MessageFilter interface, which has a single selectMessage method which is passed a TopicMessage and can return one of the following values:

- The same message as passed in – Accepting the message.
- Null – Rejecting the message and indicating that it must not be processed.
- A different message from that passed in – Replacing the message.

Replacement must be done with care as it involves creating a new message which has the same basic characteristics of the original (topic name etc).

Message filters are used to filter what is queued for a particular client – see the following section.

Client queue message filtering

You can specify message filters to be used for a specific client to filter the messages that get queued for the client.

Messages get queued for a client as a result of a publish to all clients subscribed to a topic or an explicit send to the client. However, it can be that you do not want all messages published on a topic to go to a particular client instance.

For example, you might want to send a message to a client only if a value in the message is greater than a certain amount or you might want to send messages through some timed tick rate (similar to throttling).

Such a filter can be set for a client to handle messages for a specified topic. You can to set a filter for a topic selector pattern so that it is used for any message with a topic that matches the pattern.

A filter can be set using the addQueueMessageFilter method on the client interface. Filters can be removed using removeQueueMessageFilter. See the API documentation for more details.

Filter example

The following example shows a publisher filtering mobile clients so that they only receive messages with the first user header set to "M"

```
@Override
protected void subscription(Client client,Topic topic,boolean loaded)
```

```
throws APIException {

 if (client.getConnectionType().isCategory(ConnectionCategory.MOBILE)) {
  client.addQueueMessageFilter(topic.getName(),new MobileFilter());
 }

 super.subscription(client,topic,loaded);
}

private class MobileFilter implements MessageFilter {
 @Override
 public TopicMessage selectMessage(TopicMessage message) {
  if ("M".equals(message.getHeader(0))) {
   return null;
  }
  else {
   return message;
  }
 }
}
```

The above example shows a filter being set for every topic which is efficient.

Single filter for all topics

The same can be achieved by setting a single filter for all topics when the client connects, for example

```
/**
 * @see ClientListener#clientConnected(Client)
 */
@Override
public void clientConnected(Client client) {
 if (client.getConnectionType().isCategory(ConnectionCategory.MOBILE)) {
  try {
   client.addQueueMessageFilter("//",new MobileFilter());
  }
  catch (TopicInvalidException ex) {
  }
 }
}
```

Note: When you apply filters in this way, ensure that you do not accidentally filter topics for which filtering is inappropriate.

Filter example

The following example shows a publisher filtering mobile clients so that they only receive messages with the first user header set to "M"

```
@Override
protected void subscription(Client client,Topic topic,boolean loaded)
throws APIException {

 if (client.getConnectionType().isCategory(ConnectionCategory.MOBILE)) {
  client.addQueueMessageFilter(topic.getName(),new MobileFilter());
 }

 super.subscription(client,topic,loaded);
}

private class MobileFilter implements MessageFilter {
 @Override
 public TopicMessage selectMessage(TopicMessage message) {
  if ("M".equals(message.getHeader(0))) {
```

```
    return null;
   }
   else {
    return message;
   }
  }
 }
```

The above example shows a filter being set for every topic which might not be very efficient

Single filter for all topics

The same can be achieved by setting a single filter for all topics when the client connects, for example

```
/**
 * @see ClientListener#clientConnected(Client)
 */
@Override
public void clientConnected(Client client) {
 if (client.getConnectionType().isCategory(ConnectionCategory.MOBILE)) {
  try {
   client.addQueueMessageFilter("//",new MobileFilter());
  }
  catch (TopicInvalidException ex) {
  }
 }
}
```

Note: When you apply filters in this way, ensure that you do not accidentally filter topics for which filtering is not appropriate.

Metadata

Metadata is data that describes data. In Diffusion terms it is something that defines the format of a Diffusion topic message.

Diffusion metadata is a generic mechanism for describing message formats regardless of the exact data representation. It describes a message in terms of *fields* within it and these fields can be grouped into *records* which themselves can be further subdivided into fields or records or both.

This generic representation of metadata can potentially be used for many different types of message data. For example, metadata can be used to describe a Diffusion record-based message where fields and records have the same meaning but certain constraints exist (for example, nesting of records is not permitted).

The purpose of metadata is to allow for programmatical modeling of data structures. It is used in the following ways:

- To define the layout of a record so that fields within the record can be addressed by name.
- To define the layout of the messages used by record topic data in terms of one or more record definitions.

Metadata in the Classic API

Metadata is available in .NET and Java in the Classic API.

Metadata structure

A metadata definition is made up of a hierarchy of metadata nodes (class `MNode`) with a message node (class `MMessage`) at the top. A message is defined as one or more field (class `MField`) and/or record (class `MRecord`) nodes. A record is also defined as one or more field and/or record nodes (`MMessage` is a specialization of `MRecord`). A field defines an elementary data item of a particular data type

Every node has a name which must be unique within its parent node. Every child node can represent one or more possible occurrences of that record or field within the parent. The number of possible occurrences of a node is described by its multiplicity.

The order in which nodes are defined within their parent defines the order that they appear in within a message.

Message metadata

A metadata message (class `MMessage`) is the top-level object of a metadata definition which represents a whole message layout.

Metadata messages are created using the MetadataFactory class specifying the type of topic data that the metadata describes. Even though metadata definitions are generic in form, the type of data can impose constraints on the metadata.

A message can be made up of one or more fields and/or records

Field metadata

A metadata field (class `MField`) defines an elementary data item within a message or record.

Every field has a data type which defines its actual representation within the message.

A field can have a default value specified to use when initializing message data.

A field has a multiplicity within its parent node.

Data types

The data type of a field defines its actual representation within the message. The data types available vary according to the topic data type that the metadata is defining. The same data type can result in different physical data representations for different topic data types.

The available data types are defined by the enum `MDataType` and the following are currently available:

Table 68: Data types

STRING	A character string. The initial default value for this type is a zero length string.
INTEGER_STRING	An integral number represented in the message as a character string. If a field is defined as this type, it can contain only numeric digits with an optional leading sign. By default, empty fields are not allowed. The initial default value for this type is 0.
DECIMAL_STRING	A decimal number represented in the message as a character string.

	Decimal fields have the number of places to the right of the decimal point defined by the scale, the default being 2. Such values can be parsed from a character string with any number of digits to the right of the decimal point but half up rounding is applied to achieve the target scale and output of the field is rendered with the specified scale. By default, empty fields are not allowed. The initial default value for this type is 0.00 (depending on scale). For comparison purposes the scale is ignored, a value of 1.50 is the same as 1.5.
CUSTOM_STRING	This is a special type where the behavior is delegated to a user written CustomFieldHandler. See the API documentation for more information. This type is available in all topic data types

Scales

Decimal format fields (such as DECIMAL_STRING) can have a scale which defines the number of places to the right of the decimal point.

The scale of a field is set using the setScale method. If a scale is not specified, 2 is assumed. This value is ignored for all but decimal format fields.

Default values

Every field can have a default value specified for it using the setDefaultValue method. If a value is not specified for the field, the default value is assumed to be the default for the data type.

When a message is created using metadata, default initialization applies the default values specified for each field.

Empty values

By default STRING type fields accept an empty field (that is, a zero length string) as input but other types do not. However, you can allow other field types to accept empty input also. This is done using the `setAllowsEmpty` method on the `MField` object.

Multiplicity

The multiplicity of a metadata field or record defines the number of times the corresponding data can occur within its parent.

Multiplicity (defined by the multiplicity class) is defined in terms of the minimum and maximum number of occurrences. Some data representations (such as protocol buffers) support variable numbers of nodes, whereas others (such as record data) support only a fixed number of nodes (where minimum=maximum) except in the last position.

Fixed multiplicity can be defined by a single number. For example, a multiplicity of 5 (`new Multiplicity(5)`) indicates that there must be exactly 5 occurrences of the node within its parent.

Variable multiplicity is defined in terms of a minimum value and a maximum value and is represented with the notation n..n. So a multiplicity of 1..5 (`new Multiplicity(1,5)`) indicates that there can be between 1 and 5 occurrences of the node within its parent. A special maximum value of -1 is used to represent no maximum so a multiplicity of 1..n (`new Multiplicity(1,-1)`) means that there can be any number of occurrences of the node but there must be at least one.

Optional nodes are indicated by a minimum value of 0. So 0..1 represents a node that can occur 0 or once and 0..n represents a node that can occur any number of times or not at all. A fixed multiplicity of 0 is not allowed

Creating a custom field handler

A custom field handler can be used for defining a data type other than those provided and can deal with any string data format required

The following example shows a handler used to represent a double value

```
public class DoubleFieldHandler implements CustomFieldHandler {

 public Object getInitialDefaultValue() {
  return new Double(0.0);
 }
 public Object parse(Object object) throws APIException {
  if (object==null) {
   return new Double(0.0);
  }

  try {
   return (Double.parseDouble(object.toString()));
  }
  catch (Throwable ex) {
   throw new APIException(
   "Unable to parse "+object+" as double value",
   ex);
  }
 }

 public boolean areEqual(Object source,Object target) {
  return source.equals(target);
 }
}
```

Defining record metadata

The generic metadata facility can be used to programmatically define a metadata message definition which can be used when creating records.

The following example shows message metadata being defined with a single record within it:

```
MMessage messageMetadata =
 MetadataFactory.newMetadata("MyMessage",TopicDataType.RECORD);

MRecord recordMetadata = messageMetadata.addRecord("MyRecord");
recordMetadata.addField("Name");
recordMetadata.addField("AccountNumber",MDataType.INTEGER_STRING);
recordMetadata.addField(
 "Price",
 MDataType.DECIMAL_STRING,
 new Multiplicity(4));
```

The preceding example shows a message being defined as a single record defined as a string field called “Name”, followed by an integer field called “AccountNumber”, followed by decimal field called “Price” which has a fixed multiplicity of 4 (that is, repeats exactly 4 times).

The following rules apply to record metadata:

- A message can have one or more records within it.
- Each record can have one or more fields defined for it.
- Only string data types are permitted for fields (STRING, DECIMAL_STRING, and so on).
- Custom data types can also be defined which can implement special behavior for fields.

- Repeating field definitions are permitted but only with fixed multiplicity (for example, you can have a multiplicity of 4 but not 1..4). The exception to this is the last field within the record which can have variable multiplicity (for example,. 0..n). When records are used within a message, only the last record within the message can have a repeating field at the end and the record itself cannot repeat.
- Fields can not exist directly under a metadata message, only within a record. Even though the message interface does permit messages containing only fields, a single record is implied in those cases
- Repeating record definitions are permitted within a message but only with fixed multiplicity. The exception to this is the last record within a message which can have variable multiplicity.
- Nested records are not permitted. You cannot define a record within a record.

A single metadata definition can be reused for any number of records. To allow reuse of the same definition at both the client and the publisher, encapsulate metadata definitions in their own Java classes.

Creating and populating a record with metadata

Once you have a metadata definition it can be reused to create and populate records

This example assumes the metadata has already been defined

```
Record record = new Record(recordMetadata);
record.setField("Name","John Smith");
record.setField("AccountNumber",123456);
record.setField("Price","100","98.8","95.25","90");
```

Having set up a record in this way, it can be written to a message using the putRecords method. The same record can be changed and reused many times if required.

The fields do not have to be set in any particular order and any that are not explicitly set take the default values as defined by the metadata. However, when a record is written to a message the fields are written in the order specified by the metadata.

To write more than one occurrence of the same record type to a message you put a record of that type into the message more than once.

Reading a record with metadata

You can read a record from a message using metadata

You can access the fields by name within a record if the metadata is specified in the nextRecord method.

```
Record recordIn = message.nextRecord(recordMetadata);

String name = recordIn.getField("Name");
String account = recordIn.getField("AccountNumber");
List<String> prices = recordIn.getFieldValues("Price");
```

If the record within the message does not match the metadata, an exception occurs.

Example of using record metadata

The example below shows a method that defines some message metadata to be used with record data. The message comprises two records

```
MMessage defineMyMetadata() throws APIException {
 MMessage message =
 MetadataFactory.newMetadata("MyMessage",TopicDataType.RECORD);

 MRecord record=message.addRecord("Record1");
 MField field;
 field=record.addField("Name");
```

```
 field=record.addField("AccountNumber",MDataType.INTEGER_STRING);
 field.setDefaultValue(-1);
 field=record.addField(
  "Price",
  MDataType.DECIMAL_STRING,
  new Multiplicity(2));
 field.setScale(3);

 record=message.addRecord("Record2");
 record.addField("AddressLine",new Multiplicity(5));

 return message;
}
```

Loading a metadata record

Currently metadata can only be described programmatically. It can be useful to employ the singleton pattern to encapsulate metadata definitions so that they only have to be loaded once.

The following class is an example of such a singleton

```
public class MyMetadata {

 private static MyMetadata theInstance = null;

 private HashMap<String,MMessage> theMetadata =
  new HashMap<String,MMessage>();

 private MyMetadata() throws APIException {
  loadMetadata();
 }

 private static MyMetadata instance() throws APIException {
  if (theInstance==null) {
   synchronized(MyMetadata.class) {
    if (theInstance==null) {
     theInstance=new MyMetadata();
    }
   }
  }
  return theInstance;
 }

 public static MMessage get(String name) throws APIException {
  return instance().getMetadata(name);
 }

 private MMessage getMetadata(String name) {
  return theMetadata.get(name);
 }

 private void loadMetadata() throws APIException {
  loadMessage1();
  loadMessage2();
  //... and so on
 }

 private void loadMessage1() throws APIException {
  MMessage message =
   MetadataFactory.newMetadata("Message1",TopicDataType.RECORD);

  MRecord record=message.addRecord("Record1");
  record.addField("Name");
  record.addField("AccountNumber",MDataType.INTEGER_STRING);

  record=message.addRecord("Record2");
```

```
  record.addField("AddressLine",new Multiplicity(5));

  theMetadata.put("Message1",message);
 }

 private void loadMessage2() throws APIException {
 // load another message format
 }
}
```

Retrieve a metadata definition

Metadata definitions can be obtained using the MyMetadata.get(name) method

For example:

```
MMessage metadata = MyMetadata.get("Message1");
```

Chapter 9

Event publishers (deprecated)

In this section:

- Event publishers that use the Classic API

An event publisher is a client that connects to a Diffusion server through the Classic API and publishes messages to a topic or topics.

The purpose of an event publisher is to feed data into a Diffusion server and, through topics, into publishers. Publishers can also send messages to event publishers, but high volumes of message traffic in this direction is not expected

> **Note:** The event publisher APIs in the Classic API are deprecated. You can use the Unified API to write a control client that has the capabilities of an event publisher.

Related Links

Unified API on page 240
The Diffusion Unified application programming interface (API) provides a consistent interface to the Diffusion server, whichever role your client application performs: standard client or control client.

Control client on page 226
A Diffusion client application can take on the role of control client.

Classic APIs on page 342
Diffusion provides a number of Application Programming Interfaces (APIs) which allow user-written applications to make use of Diffusion.

Event publishers that use the Classic API

A publisher receives messages from event publishers through the publisher's `messageFromEventPublisher` method. The message (a `TopicMessage`) and a reference to the event publisher connection (`EventConnection`) are passed. A publisher can receive messages from more than one event publisher connection – this is not typical.

Receiving event publisher notifications

A publisher can receive notifications when an event publisher connects or disconnects. This can be important because the publisher might rely upon the event publisher for the integrity of its data. To receive such notifications the publisher adds a listener using the `Publishers.addEventListener` method.

Sending messages to event publishers

Messages can be sent from publishers to event publishers using the send method on the `EventConnection` interface. Only delta messages can be sent.

Event publishers receiving messages from publishers

Event publisher applications receive messages from publishers on their listener interface.

For example, in the Java API, such a message is received on the `EventPublisherListener.messageFromServer` method.

Configuring event publishers

Event publishers are configured programmatically by setting values of the event publisher connection.

Each event publisher has one or more `EventPublisherConnection` objects that it uses to communicate with a publisher. Each of these connections can be configured separately. An `EventPublisherConnection` is configured programmatically through the Java API.

There are two objects that can be configured for the event publisher, the `EventPublisherConnection` and the `ServerDetails`.

The `EventPublisherConnection` object enables you to configure the following values:

- connection host
- connection port
- message queue size
- topic aliasing
- a listener

- The host and port are used to identify the Diffusion server, these can also be configured in an `ServerDetails` object.
- The message queue size is the number of messages that can be queued on the event publisher side to be sent to the publisher.
- Topic aliasing can be enabled or disabled for use with the event publisher.
- The listener allows the event publisher to handle disconnections and messages from the publisher.

The `ServerDetails` object enables you to configure the following values:

connection host and connection port	The host and port are used to identify the Diffusion server, these can also be configured in an `EventPublisherConnection` object.

auto-acknowledgments

The auto-acknowledgments allows messages that require acknowledgment to be acknowledged automatically.

connection timeout

The connection timeout refers to how much time passes before the event publisher accepts that it cannot open a connection.

credentials

The credentials that are used to authenticate a connection can be configured.

input and output buffer sizes

The input and output buffer sizes are the size of the read/write socket buffers.

proxy

The proxy allows connections to go through a proxy server.

SSL content

The SSL context is used to set up secure connections.

topics

The topics that are subscribed to automatically on connection.

write timeout

The write timeout is how much time passes before the event publisher accepts that it cannot write to the connection.

Further details of the `EventPublisherConnection` and the `ServerDetails` can be found in the API documentation for these classes.

Tip: You can use the features in the Unified API to perform some of the actions of an event publisher. For more information, see Control client on page 226.

Chapter 10

Control client

In this section:

- Advantages of control client
- Developing a control client
- Event handling with a control client
- Maintaining topics from a control client
- Updating topics from a control client
- Managing subscriptions from a control client
- Managing clients from a control client
- Messaging from a control client
- Comparison of remote control and control client

A Diffusion client application can take on the role of control client.

Control clients can perform the following actions:

- Define and create data structures, such as the layout of the topic tree and metadata that describes the format of messages published to topics
- Publish data to topics
- Send messages to specific clients
- Handle events that occur at server level, on topics, or on parts of the topic tree. For example:
  - At server level, client authentication requests
  - At topic level, subscription requests

Control clients interact with the Diffusion server through the Unified API and communicate over any of the supported protocols.

If you want to use control features in your client application, you must use the Unified API. However, other kinds of client can still connect to a version 5.1 Diffusion server through the Classic API.

Related Links

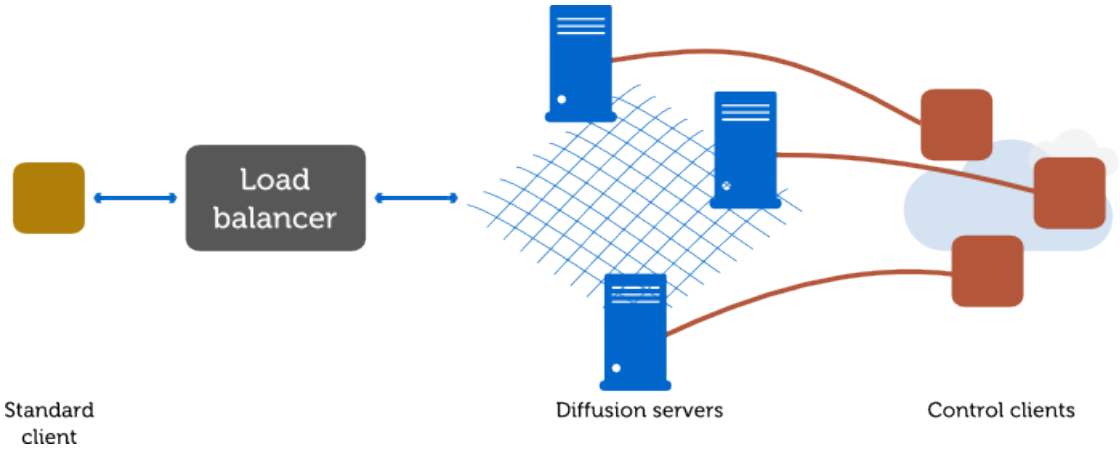
Unified API on page 240

The Diffusion Unified application programming interface (API) provides a consistent interface to the Diffusion server, whichever role your client application performs: standard client or control client.

Advantages of control client

Diffusion version 5.0 introduces the control client, a client role that uses the Unified API to perform control actions on the Diffusion server.

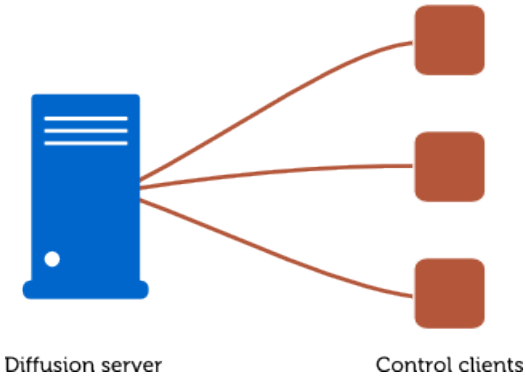
Abstracting your control components



Using a control client enables you to locate your control components outside of the Diffusion server. You can deploy your control clients on separate systems to the Diffusion servers, in a separate data center, or in the cloud, enabling greater flexibility in your architecture.

Keeping your control components abstracted from Diffusion enables you to treat Diffusion as a messaging fabric. This supports scalable and cloud deployments.

Load balancing



Multiple control clients can connect to a single Diffusion server. If these control clients have registered handlers on the same events at the server, the server can load balance requests between these control clients.

Requests are load balanced between control clients first by using a sticky-by-client approach – if a control client has already handled a request from a standard client, further requests from the standard client are routed to that control client – and secondly using a round-robin approach – if no requests have previously been received from a standard client, the standard client's first request is routed to the next control client in the round robin.

Using your organization's preferred development language

Control clients can be written in one of a number of languages. You can use your organization's preferred language to implement your control clients and seamlessly integrate with your existing systems. The following languages are supported in version 5.1 for production use:

- Java

The following languages have a limited feature set and are supported for development use only:

- .NET
- C

Control client can use any of the supported transport protocols to communicate with Diffusion and can take advantage of cryptographic protocols such as SSL to ensure secure communication. Diffusion authenticates all connections from clients.

Related Links

Unified API on page 240
The Diffusion Unified application programming interface (API) provides a consistent interface to the Diffusion server, whichever role your client application performs: standard client or control client.

Developing a control client

The control client features are available through the Unified API. You can develop a client that uses these features to create and manage topics and subscriptions and to authenticate and manage other clients.

Before you begin

Ensure that your Diffusion server is configured to accept connections from clients that use the Unified API. You can configure this in the `Connectors.xml` configuration file. For more information, see Connectors.

Procedure

1. Use the client libraries and Unified API provided with Diffusion to develop a client application that opens a session with the server.

   For more information, see the examples in the Unified API on page 240 section.

   - Getting started with the Java API on page 260
   - Getting started with the .NET API on page 263
   - Getting started with the C API on page 264
2. Use the session to get the control features that you require.
   - Use the AuthenticationControl feature to enable a control client session to authenticate other clients.

     For more information, see AuthenticationControl on page 267 and Authentication handlers on page 418.
   - Use the ClientControl feature to enable a control client session to receive notifications about other clients and to manage other clients. For example, to enable throttling for a client if the client's queue is too deep.

     For more information, see ClientControl on page 274.
   - Use the TopicControl feature to enable a control client session to create and manage topics and to provide state for delegated topics.

     For more information, see TopicControl on page 297.
   - Use the TopicUpdateControl feature to enable a control client session to update topics.

     For more information, see TopicUpdateControl on page 325 and Updating topics from a control client on page 234.
   - Use the SubscriptionControl feature to enable a control client to subscribe other clients to topics and handle routing topic subscription requests.

     For more information, see SubscriptionControl on page 294.
   - Use the MessagingControl feature to enable a control client to send messages to specific client sessions and receive messages sent by clients to topics.

     For more information, see MessagingControl on page 283.

3. Develop your required control logic using the classes and interfaces provided by the Unified API.
4. Compile and run your control client.
5. Optional: You can run multiple instances of your control client that all connect to the same server and register the same handlers.

   This enables the server to load balance its requests across multiple control clients and provides redundancy.

Related Links

Unified API on page 240
The Diffusion Unified application programming interface (API) provides a consistent interface to the Diffusion server, whichever role your client application performs: standard client or control client.

Event handling with a control client

A control client application can handle events that occur at a server level, on topics, or on parts of the topic tree.

Server-level handlers

When a Diffusion server receives a request for a server-level action, it checks which control client has registered a handler for the requested action. For example, a client authentication request.

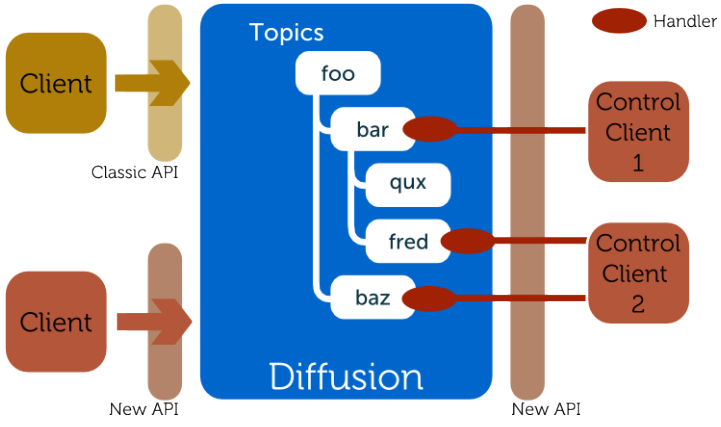
In the case where two or more control clients have registered handlers on the same action, the server chooses which control client to pass the request to based on the following criteria:

- The control client that has previously handled requests from this client

  This allows the control client to access information it might have cached about its interactions with the client and work more efficiently.
- To balance the load between the available control clients

  If the requesting client has not previously made any requests, this request goes to a control client chosen by the round-robin approach.

Topic-level handlers



When a server receives a request for an action on a topic or topic tree, it checks which control client has registered a handler on that topic or that part of the topic tree for the requested action.

If more than one control client has registered a handler for that topic and action, the server chooses which control client to pass the request to based on the following criteria:

- The most specific topic path

For example, if Control Client 1 is registered to handle subscribe actions on the topic /foo/bar and all of its subtopics and Control Client 2 is registered to handle subscribe actions on the topic /foo/bar/fred, the server passes a request on /foo/bar/fred to Control Client 2.

In the case where two or more control clients have registered handlers on the same topic and action, at the same level of specificity, the server chooses which control client to pass the request to based on the following criteria:

- The control client that has previously handled requests from this client

  This allows the control client to access information it might have cached about its interactions with the client and work more efficiently.
- To balance the load between the available control clients

  If the requesting client has not previously made any requests, this request goes to a control client chosen by the round-robin approach.

> **Note:** If the object registered against a topic or branch of the topic tree is an update source, different criteria are used. Only one update source can be the active update source for a topic. The active update source updates the topics in that branch of the topic tree. The other update sources for that branch of the topic tree remain in standby and do not update the topics.

Maintaining topics from a control client

A control client can use the TopicControl feature of the Unified API to add and remove topics at the server.

A control client can create any type of topic.

Currently all topics created using a control client have a lifespan the same as the server. The topics remain at the server even after the control client session that created them has closed.

Adding new topics

For a control client to create a new topic it must first define the topic details that describe the topic. Builders of topic details can be created using the TopicControl feature. For more information, see Topic details on page 255.

You can use the same instance of topic details to create many topics. This is recommended when many topics with the same definition are to be created, because caching optimizations occur that prevent complex definitions from being transmitted to the server many times.

For some types of topic, setting up metadata is part of the task of describing the topic.

The control client can use the TopicControl feature to supply the initial state of the topic to the server, as content, when the topic is created.

The `addTopic` operation is asynchronous and calls back to notify of either successful creation of the topic or failure to create the topic. If the topic add fails at the server, the reason for failure is returned. Possible reasons for failure include the following:

- The topic already exists at the server with exactly the same details
- The topic already exists at the server
- The name of the supplied topic is not valid
- The supplied details are not valid. This can occur only if properties are supplied.
- A supplied user class cannot be found or instantiated
- A referenced topic cannot be found
- Permission to create the topic was denied
- An error occurred trying to initialize the newly created topic with the supplied content, possibly because it was not validly formatted

A control client can create topics subordinate to topics created by another control client.

> **Note:** It is not currently possible to add new topics under branches of the topic tree that have been created by internal publishers.

Dynamically adding topics

You can use the TopicControl feature to dynamically create topics on demand when a client tries to subscribe or fetch a topic that does not exist.

The control client can register itself as a handler for missing topics for any part of the topic tree. The control client is notified of attempts to subscribe to or fetch topics that are subordinate to that topic and that do not exist. This enables the control client to create the topics and notify the server that the client operation subscribe or fetch can proceed.

> **Note:** The handler is called only when a client attempts to subscribe or fetch using a single topic path. If another type of selector is used to subscribe to or fetch the topic, the handler is not notified.

Providing state for delegated topics

A control client can register itself as a state provider for any branch of the topic tree.

When a control client is registered as a state provider, all requests for the state of delegated topics in that branch are routed to the control client, which returns content that defines the current topic state.

Removing topics

A control client can remove topics anywhere in the topic tree. The remove operation takes a topic selector, which enables the control client to remove many topics at once. The removal of a topic also causes the removal of all topics beneath it in the tree.

Related Links

TopicControl on page 297
Use the TopicControl feature to enable a control client session to create and manage topics and to provide state for delegated topics.

Creating paged topics

Use the `TopicDetails.Builder` classes with the TopicControl feature to create paged topics.

You can create a paged topic from the control client by using the following topic details builder interfaces are provided to create paged topics:

`PagedRecordTopicDetails.Builder`
: Use this builder to create a paged record topic.

`PagedStringTopicDetails.Builder`
: Use this builder to create a paged string topic.

Paged topics can be ordered or unordered. You can define the ordering of a paged topic by using a user-defined comparator class that is located on the Diffusion server or by declaring the rules that are used for the ordering when you create the topic.

Creating an unordered paged topic

Use the `unordered` method of the topic details builder to set the paged topic to be unordered. By default paged topics are unordered, but you can also use this method to remove any ordering that had previously been specified for the paged topic.

In an unordered paged topic, the topic updates are added to the end of the paged topic in the order they are received.

Creating an ordered paged topic that uses a comparator

Use the `order` method of the topic details builder to set the paged topic to be ordered.

The `order` method can take the fully qualified name of a comparator class that is located on the class path of the Diffusion server, for example `com.example.comparators.MyComparator`. The comparator class must implement `java.util.Comparator<T>`, where *T* is either `Record` or `String` depending on the type of paged topic you are creating.

Creating an ordered paged record topic that uses declared rules

Use the `order` method of the paged record topic details builder to set the paged record topic to be ordered by one or more of its fields.

The `order` method can take one or more `OrderKey` objects.

An order key specifies the name of the field to order the lines in the paged record topic by. Because the order key uses a field name, you must define the fields that are in the paged record topic before you can specify which of these fields to use for ordering.

An order key can also, optionally, include an `Order` instance and a `Rules` instance. The `Order` instance defines the direction in which the lines are sorted and can be either DESCENDING or ACSENDING. If no order is defined, the default is ASCENDING. The `Rules` instance defines what additional ordering rules are applied to the ordering. The `Rules` instance can be of the type NO_RULES or the type COLLATION. If no rules are defined, the default is NO_RULES. You can use COLLATION-type rules to specify additional ordering rules by creating a `CollationRules` instance that defines these rules. The rules are defined using a String that uses the format defined by the Diffusion `java.text.RuleBasedCollator`. For more information, see RuleBasedCollator.

Supply the order keys to the `order` method in the order that they are to be applied. For example, if you specify an order key for the "Name" field then an order key for the "Date" field, lines are ordered first by the "Name" field and then, within those lines that have the same value in the "Name" field, by the value in the "Date" field.

Creating an ordered paged string topic that uses declared rules

Use the `order` method of the paged string topic details builder to set the paged string topic to be ordered.

The `order` method can take an `Order` instance and a `Rules` instance. If no parameters are passed to the `order` method, default values are used. The `Order` instance defines the direction in which the lines are sorted and can be either DESCENDING or ACSENDING. If no order is defined, the default is ASCENDING. The `Rules` instance defines what additional ordering rules are applied to the ordering. The `Rules` instance can be of the type NO_RULES or the type COLLATION. If no rules are defined, the default is NO_RULES. You can use COLLATION-type rules to specify additional ordering rules by creating a `CollationRules` instance that defines these rules. The rules are defined using a String that uses the format defined by the Diffusion `java.text.RuleBasedCollator`. For more information, see RuleBasedCollator.

Duplicate lines

You can also pass a duplicates policy into the `order` method, which defines where duplicate lines are inserted into the paged topic. The duplicates policy can have one of the following values:

FIRST
: Insert the new line before the first matching duplicate.

LAST
: Insert the new line after the last matching duplicate.

NOT_ALLOWED
: Do not insert the duplicate and return an error.

If no duplicates policy is defined for an ordered paged topic, the default is NOT_ALLOWED and an error is returned if an attempt is made to add a duplicate line.

Related Links

Managing paged topic data in the Unified API on page 167
You can use the Unified API to create and update paged topics.

TopicControl on page 297
Use the TopicControl feature to enable a control client session to create and manage topics and to provide state for delegated topics.

Removing topics with sessions

A control client can use the TopicControl feature of the Unified API to specify that the Diffusion server removes a topic or topics after the control client session closes or fails.

When a control client registers a request that a branch of the topic tree be removed when its session closes the following events occur:

1. The control client registers the removal request on a branch of the topic tree and passes in a topic tree handler.
   - The removal request is registered against a topic path. This is a path that identifies a branch of the topic tree, for example foo/bar. The removal request is registered for the branch of the topic tree, for example the topics foo/bar/baz and foo/bar/fred/qux are included in the specified branch of the topic tree.
   - You cannot register a removal request above or below an existing removal request. For example, if a control client has registered a removal request against foo/bar/fred another control client cannot register a removal request against foo/bar or foo/bar/fred/qux.
2. The server validates the request and gives one of the following responses:
   - If the request is not valid, the server calls the `onError` callback of the topic tree handler.

     For example, a registration request is not valid if it registers against a topic branch that is above or below a branch where an existing removal request is registered.
   - If the request is valid, the server calls the `OnActive` callback of the topic tree handler and provides a `Registration` object to the control client.
3. If the control client wants to deregister a removal request, it can call the `onClose` method of the `Registration` object for that removal request.
4. Other control clients can register removal requests against a topic that already has a removal request registered against it. For example, if one session on the server has a removal request registered against foo/bar/baz, another session on that server can also register a removal request against foo/bar/baz.
5. When a control client session closes or fails, if it has registered removal requests one of the following things happens:
   - If there are still open sessions that have removal requests for that branch of the topic tree, the Diffusion server takes no action.
   - If there are no open sessions that have removal requests for that branch of the topic tree, the Diffusion server removes all topics in that branch of the topic tree.

Removal requests and topic replication

If all sessions on a Diffusion server that have a remove topics removal request for a branch of the topic tree close, the topics are removed even if that topic is replicated and sessions on other Diffusion servers have removal requests registered against that part of the tree. When the topics are removed on the server, that change is replicated to all other servers that participate in replication for these topics.

Related Links

TopicControl on page 297

Use the TopicControl feature to enable a control client session to create and manage topics and to provide state for delegated topics.

Updating topics from a control client

A control client can use the TopicUpdateControl feature of the Unified API to update topics.

To update a topic a control client must register with the Diffusion server as an update source for the branch of the topic tree that contains the topic to be updated. The server passes the control client an updater that the control client can use to update the topic.

Registering as an update source

A control client must register as an update source for a branch of the topic tree to be able to publish content to topics in that branch. When a control client registers as an update source the following events occur:

1. The control client requests to register an update source on a branch of the topic tree.
   - The update source is registered against a topic path. This is a path that identifies a branch of the topic tree, for example foo/bar. The update source is registered as a source for that branch of the topic tree, for example the topics foo/bar/baz and foo/bar/fred/qux are included in the specified branch of the topic tree.
   - You cannot register an update source above or below an existing update source. For example, if a control client has registered an update source against foo/bar/fred another control client cannot register an update source against foo/bar or foo/bar/fred/qux.
   - You can register an update source against a topic owned by an existing publisher or a topic that has an update source created by the server that is used for topic failover.
2. The server validates the registration request and returns one of the following responses:
   - If the request is valid, the server calls the `OnRegister` callback of the update source and passes a `RegisteredHandler` that you can use to unregister the update source.
   - If the request is not valid, the server calls the `onClose` callback of the update source.

     For example, a registration request is not valid if it registers against a topic branch that is above or below a branch where an existing update source is registered.
3. When the update source is registered, the server calls one of the following callbacks:
   - If the update source is the primary update source, the server calls the `onActive` callback of the update source.
   - If another update source is already the primary source for this branch of the topic tree, the server calls the `onStandby` callback of the update source.
4. If an update source is on standby, the update source cannot update the topics it is registered against. A standby update source can, in future, become active and become the primary update source for a branch of the topic tree.
5. If an update source is active, the server provides the update source with an `Updater`. The update source can use the `Updater` to update the topics it is registered against.

Using an updater to publish content to topics

The control client that is the active update source for a branch of the topic tree has an updater that it can use to publish content to topics in that branch. When the control client uses an updater method to publish content, it passes in the following parameters:

Topic path

The path to the topic to be updated. This topic must be in the branch of the topic tree that the control client is the active update source for and that the updater is associated with.

Content

The information about the update can be provided as either a simple `Content` object or as a more complex `Update` object.

The content that is to be published to the topic. The client must use the appropriate content type when formatting the content. If the content uses the wrong content type for the topic, it can cause an error.

Update

The information about the update can be provided as either a simple `Content` object or as a more complex `Update` object.

An update that contains the content that is to be published to the topic and other information about the update, such as its type.

Use the `Builder` methods provided in the Unified API to build your `Update` objects.

Context

A context object can be passed in to the update method that provides application state information.

Callback

The server uses the callback to return the result of the update. If the update completes successfully, the server calls the callback's `onComplete` method. Otherwise, the server calls the callback's `onError` method.

Related Links

Failover of active update sources on page 70
You can use failover of active update sources to ensure that when a server that is the active update source for a section of the topic tree becomes unavailable, another server is assigned to be the active update source for that section of the topic tree. Failover of active update sources is enabled for any sections of the topic tree that have topic replication enabled.

Failover of active update sources on page 70
You can use failover of active update sources to ensure that when a server that is the active update source for a section of the topic tree becomes unavailable, another server is assigned to be the active update source for that section of the topic tree. Failover of active update sources is enabled for any sections of the topic tree that have topic replication enabled.

TopicUpdateControl on page 325
Use the TopicUpdateControl feature to enable a control client session to update topics.

Building updates for paged topics

Use the `UpdateFactory` classes with the TopicUpdateControl feature to create the `Update` objects that can be used to update paged topics.

You can update a paged topic from the control client in the same way as you update any other topic: by registering as an update source for the topic and using an updater to publish content to the topic. When updating a paged topic, you must pass an `Update` object to the updater. You can use update factories to create the `Update` object. The following update factory interfaces are provided for use with paged topics:

`PagedRecordOrderedUpdateFactory`

For use when the topic is a paged record topic and has an ordering policy of declared or comparator.

`PagedRecordUnorderedUpdateFactory`
: For use when the topic is a paged record topic and has an ordering policy of unordered.

`PagedStringOrderedUpdateFactory`
: For use when the topic is a paged string topic and has an ordering policy of declared or comparator.

`PagedStringUnorderedUpdateFactory`
: For use when the topic is a paged string topic and has an ordering policy of unordered.

Ensure that you use the appropriate update factory for the type of paged topic you are updating.

Building updates for ordered topics

When building an update for an ordered topic, you can instruct the Diffusion server to add a line, remove a line, or update a line in the topic. The server uses the ordering policy and duplicates policy of the paged topic to determine where the line is added, whether the line is added, or which line is removed or updated.

Building updates for unordered topics

When building an update for an unordered topic, you can instruct the Diffusion server to add one or more lines, insert one or more lines at a specified index, update the line at a specified index, or remove one or more lines starting at a specified index.

Related Links

Managing paged topic data in the Unified API on page 167
You can use the Unified API to create and update paged topics.

TopicUpdateControl on page 325
Use the TopicUpdateControl feature to enable a control client session to update topics.

Managing subscriptions from a control client

A control client can use the SubscriptionControl feature of the Unified API to subscribe other client sessions to topics that they have not requested subscription to themselves and also to unsubscribe clients from topics. It also enables the control client to register as the handler for routing topic subscriptions.

Subscribing and Unsubscribing Clients

A control client can subscribe clients that it knows about to topics that those clients have not explicitly requested. It can also unsubscribe clients from topics.

A session identifier is required to specify the client session that is to be subscribed or unsubscribed. Use the ClientControl feature to get the identifiers for connected client sessions.

The SubscriptionControl feature uses topic selectors to specify topics for subscription and unsubscription. Many topics can be specified in a single operation.

Acting as a Routing Subscription Handler

Routing topics can be created with a server-side handler that assigns clients to real topics. However, you can omit the server-side handler such that subscriptions to routing topics are directed at a control client acting as a routing subscription handler.

A control client can register a routing subscription handler for a branch of the topic tree. Any subscription requests to routing topics in that branch that do not have server-side handlers) are passed to the control client for action.

On receipt of a routing subscription request the control client can respond with a "route" request that specifies the path of the actual topic that the routing topic maps to for the client. This subscription fails if the target topic does not already exist.

The control client can complete other actions before calling back to "route". For example, it could use the TopicControl feature to create the topic that the client is to map to.

Alternatively, the control client can "defer" the routing subscription request in which case the client remains unsubscribed. This is similar to denying it from an authorization point of view.

The control client must reply with a "route" or "defer" for all routing requests.

Related Links

SubscriptionControl on page 294
Use the SubscriptionControl feature to enable a control client to subscribe other clients to topics and handle routing topic subscription requests.

Managing clients from a control client

A control client can use the ClientControl feature of the Unified API to manage other client sessions.

Receiving notifications of other client sessions

A control client can set a `SessionDetailsListener` that is notified of all client sessions that open and close.

When the control client first registers a listener, it receives a notification for every client session that is currently open.

When the control client declares a listener, it can specify exactly the level of detail that the it wants to receive for each client session. The control client can request no details, in which case the control client receives only session identifiers.

When a control client is notified of a session closing, it also receives the reason that the session was closed.

Handling client queue events

A control client can register a `QueueEventHandler` that is notified when outbound client queues at the server reach pre-configured thresholds.

A control client can respond to a client queue getting full by setting conflation on for the client. A control client is also able to set throttling on for specific clients, which also sets conflation.

Getting details of specific clients

A control client can request details of any client session from the server.

Closing client sessions

A control session can close any client session.

Related Links

ClientControl on page 274

Use the ClientControl feature to enable a control client session to receive notifications about other clients and to manage other clients.

Messaging from a control client

A control client can use the MessagingControl feature of the Unified API to send individual messages to any known client on any topic. It can also register a handler for messages sent from clients.

Sending messages to clients

A control client can send a message to any known client session on any topic, regardless of the topic type. The messages are delivered to standard clients that use the Unified API through the Messaging feature or to clients that use the Classic API through the topic listener mechanism.

The control client requires the session identifier of the client session it is sending to. Use the ClientControl feature to get the identifiers for connected client sessions. A control client can also send messages back to clients from which it receives messages through a *message handler*.

The body of the message that is sent is represented as *content*. Any of the content builder features can be used to build message content. With messaging you can also send empty content.

When sending a message certain additional options can also be specified:

Headers
: A set of string values that can be sent along with the content.

Priority
: Use this to specify the priority used when queuing the message for the client at the server.

Registering message handlers

A control client can add a message handler for any branch of the topic tree. This handler receives messages sent from clients on any topic beneath that branch unless overridden by a handler registered for a more specific branch.

Each control session can register only a single handler for any branch in the topic tree. To change the handler for a particular branch, the previous handler must be closed.

A message received by a message handler comprises content and message context which can include headers. The content can be empty. The application interprets the content of messages. Message content is not required to match the content definition of the topic that the message is sent on.

Related Links

MessagingControl on page 283
Use the MessagingControl feature to enable a control client to send messages to specific client sessions and receive messages sent by clients to topics.

Comparison of remote control and control client

Remote control is the control mechanism used in Diffusion version 4. Diffusion version 5.0 introduced the control client, a client role that uses the Unified API to perform the equivalent functions to remote control.

The control client functionality replaces remote control. Control client and remote control differ in the following ways:

- Control client uses the Unified API and remote control used the Classic API. There are no longer any control features available through the Classic API. If you want to use control features with Diffusion version 5.1, you must implement a control client.
- Control client is more secure than remote control. The control client uses the client protocols to connect to Diffusion and communicates using the Unified API. This enables the control client to use all of the security features of a true client application.
- There is no concept of domains or topic ownership for control clients. A remote control handled requests only on topics that were subtopics of its domain topic, which were created and owned by that remote control. A control client can register a handler on any topic regardless of where it is in the topic tree or what the topic was created by.
- There is no requirement for a publisher to exist on the Diffusion server. For remote control, a publisher had to exist on the Diffusion server and that publisher had to be enabled for remote control by creating a topic to handle incoming requests and responses from the remote controllers. Instead control clients communicate directly with the Diffusion server.
- The processing of topic selectors used for subscription is done on the server. For remote control, the server passed through the whole topic selector and the remote control was required to do the processing necessary to match the selector to topics. From Diffusion version 5.0 onwards, topic selectors are interpreted on the server and a request is sent to the appropriate control client for each topic that matches the topic selector.
- Control client can use any of the supported protocols to communicate with the Diffusion server. Remote control was restricted to using only DPT.
- All of the control client's interactions are asynchronous. This reduces the likelihood of blocking on calls.
- You can also use the control features in the Unified API to include event publisher capabilities in your control client.

Related Links

Unified API on page 240
The Diffusion Unified application programming interface (API) provides a consistent interface to the Diffusion server, whichever role your client application performs: standard client or control client.

Chapter 11

Unified API

In this section:

- Advantages of the Unified API
- Key concepts in the Unified API
- Supported client platforms
- Features

The Diffusion Unified application programming interface (API) provides a consistent interface to the Diffusion server, whichever role your client application performs: standard client or control client.

The Unified API files and associated client library files are located in the `clients` directory of your Diffusion installation.

> **Note:** In Diffusion 5.1, only the Unified API supports control use cases. Remote control functionality has been removed from the Classic API. You can still develop your standard clients using the Classic API.

Throughout the Diffusion 5.x release series, the Unified API will introduce all the features previously provided by the Classic API. Those features included in the Unified API in 5.1 are the following:

- Non-control features
  - Topics
  - Messaging
  - Security
  - Pings
- Server-level control features
  - AuthenticationControl
  - ClientControl
- Topic-level control features
  - TopicControl
  - TopicUpdateControl
  - SubscriptionControl
  - MessagingControl

If you write a client using the Unified API, ensure that you configure connectors on your Diffusion server to accept connections from clients that use the Unified API. For more information, see Connectors

By default, clients that connect using the Unified API connect on port 8081.

Related Links

Diffusion APIs on page 42

Diffusion provides application programming interfaces (APIs) that you can use to create applications that interact with the Diffusion server.

Control client on page 226
A Diffusion client application can take on the role of control client.

Classic APIs on page 342
Diffusion provides a number of Application Programming Interfaces (APIs) which allow user-written applications to make use of Diffusion.

Advantages of the Unified API

The Unified API available in version 5.0 and later provides a different approach to that of the Classic API.

Consistent

The Unified API provides a single consistent interface to communicate with the Diffusion server, whatever the role of your client application. In the Classic API, separate interfaces were provided for the different client roles: standard client, control client, and event publisher.

Modular

Interfaces are provided on a feature-by-feature basis. There is a clear delineation between features. At runtime, the client starts only those services that it uses. Later versions of Diffusion can provide additional features without requiring that you recompile your existing clients.

By contrast, the Classic API had a monolithic structure.

Asynchronous

All calls in the Unified API are asynchronous. Asynchronous calls remove the possibility of your client becoming blocked on a call. The Unified API also provides context-specific callbacks, enabling you to pass contextual information with a callback, and a wider range of event notifications than was available in the Classic API.

Independent

The Unified API is provided in library files independent of a Diffusion server installation. This enables you to develop against the interfaces without requiring a Diffusion server and license on your development system. The Classic API is available only as part of the Diffusion jar file.

Session-oriented

The Unified API introduces the concept of a session. A session can survive the temporary loss of connection to the server. Clients are notified when the connection is lost.

Future releases of Diffusion will automate reconnection, queue, and replay options, and will increase the resilience of sessions.

Key concepts in the Unified API

The following section describes some of the key concepts associated with the Unified API.

Sessions

A session represents a logical context between a client and one or more Diffusion servers.

Typically, a session represents a single connection to a single server. However, in the event of connection failure the session can automatically reconnect to the same server or even to failover to another server and still retain its context. For more information, see Session replication on page 67

You can open a session by using a session factory and specifying the URL of the Diffusion server. Diffusion URLs take the following form:

```
scheme://host:port
```

scheme

The transport used to make the connection. For example, `dpt`, `dpts`, `http`, `https`, `ws`, or `wss`.

host

The host name or IP address of the system on which the Diffusion server is located.

port

The port on which the Diffusion server has a connector that accepts connections from clients using the Unified API. You can configure connectors in the `Connectors.xml` configuration file of the Diffusion server.

The act of opening the session establishes a connection with the server.

The session does not receive input from the server until it is started, but can be used to obtain features and perform certain setup actions before it is started.

You can use the session factory to specify certain session properties, including the following:

Error handler

A callback for session errors. You must implement this callback to act upon session errors. Errors that can be handled by the application are typically returned through feature callbacks. The normal action for a session error is to close the session.

Listener

An optional callback that can be used to notify the client of state changes on the session.

Principal

The security principal. By default, a session uses a zero length string, "", as its principal. This value is interpreted as "anonymous".

Credentials

The credentials associated with the principal that enable the server to authenticate the principal.

Buffer sizes

The input and output buffer sizes. To gain optimum performance, configure these values to match the server values. The buffers must be large enough to handle the largest single content item expected.

SSL context

The SSL context if a secure connection is required.

When a session is opened it is assigned a unique session identifier by the server which identifies the session even if it becomes connected to another server.

The following diagram shows the session state model:

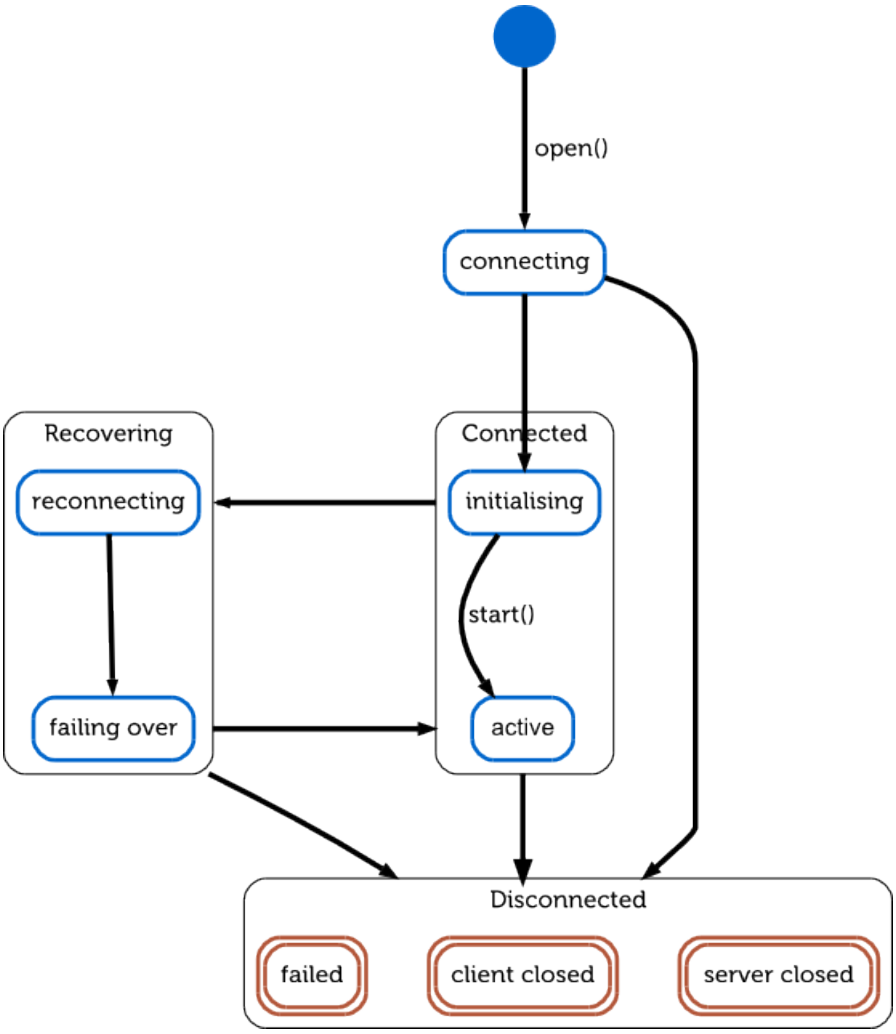


Figure 21: Session state model

Features

Units of functionality within the Unified API are exposed as features.

You can obtain features from the session. You can have only one instance of each available feature per session. Each feature is dynamically instantiated only when it is first requested from the session. It can be efficient for a session to obtain references to the features it uses before starting the session.

Standard features are available to all sessions and enable clients to subscribe to topics to receive updates or to send and receive messages or both.

Control features are for control clients only and enable the client to perform actions upon the server similar to those that can be achieved from within an internal publisher. For example, topics can be created, updated, and deleted. A control client can also monitor and manage other client sessions.

Related Links

Features on page 266

The Unified API is organized into features that make up conceptual groupings of capabilities.

Callbacks and streams

The Unified API is asynchronous and uses callbacks to notify the client of successful or failed calls. The following types of callback exist: callback, stream, contextual callback, and contextual stream.

Callbacks

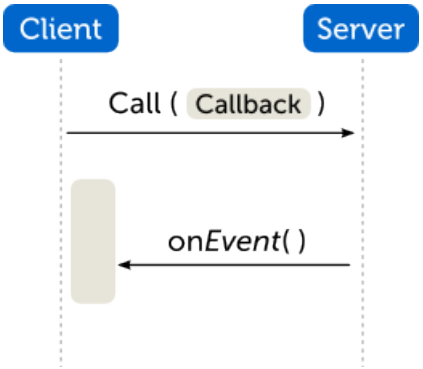


Figure 22: A callback

A callback is a response object that the client passes to an asynchronous call. When the call completes its actions it can use the callback object to respond to the client by calling a method on the callback. A callback can receive only one response from a call.

All callbacks in the Unified API have the following methods:

`onEvent`

Where *`Event`* depends on the specific callback and describes the event that occurs when the call has completed. This method is used to provide the response from the call.

`onDiscard`

This method is used to indicate that the call has not returned a response, but that the callback is no longer required. This can happen when a call times out or a session is closed.

Streams

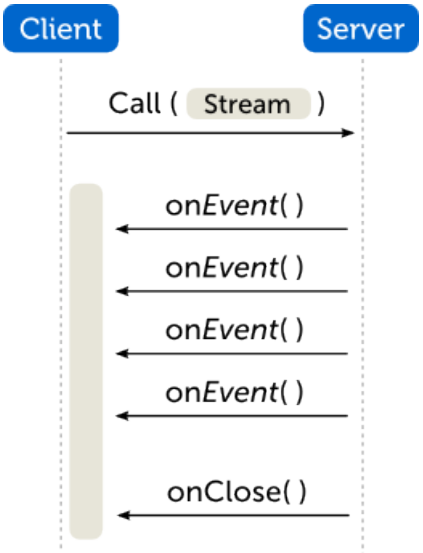


Figure 23: A stream

A stream is a response object that the client passes to an asynchronous call. When the call is ready to respond it calls a method on the stream to respond to the client. A stream differs from a callback, because it can receive more than one response. A stream continues to receive responses until it is closed or discarded.

All streams in the Unified API have the following methods:

`onEvent`

Where *`Event`* depends on the specific stream and describes the event that occurs when the call is ready to return a response. This method is used to provide the responses from the call.

`onClose`

This method is used to indicate to the stream that all responses have been received from the call. After this method is called, the stream can not receive any more responses.

`onDiscard`

This method is used to indicate that the call has not returned a response, but that the callback is no longer required. This can happen when a call times out or a session is closed.

Contextual callbacks and streams

A contextual callback or contextual stream works in the same way as a callback or stream, except that when it is created an object is associated with it that provides application state information. For example, when a client requests information to populate a UI element it can associate an object that represents the UI element with the callback for the request. When the callback is called and provided with the information requested, it can use the associated context object to complete the appropriate action with that information: in this case, populating the UI element with the information.

Handlers and listeners

Handlers are part of the Diffusion interface that represent a server side presence for a particular session. Listeners are used simply to receive notifications from the server and do not imply any active server side presence on behalf of the client.

Server handlers

Server handlers are components that a control client application registers as a handler of notifications to the client application from a server-side presence registered through the client session.

The following uses are examples of the use of server handlers in the control client:

- To receive notifications of all clients that connect to or disconnect from the server.
- To authenticate other client sessions.
- To receive notifications regarding significant events on the queues of other clients.

A client is able to register only one server handler of any particular type.

All server handlers have an `onActive` callback that is used to pass a registered handler reference, which the application can use to close the registration when required. There is also an `onClose` callback, which is used to notify that the registration has been closed.

Topic tree handlers

These are components that a control client registers to handle events from the server relating to a particular part of the topic tree. Typically, when a handler is registered for a topic within the topic tree then the handler also applies to all topics beneath it within the topic tree unless superseded by another handler further down the tree.

What happens when more than one session registers a handler for the same part of the topic tree would depend upon the feature with which it is being used.

The following uses are examples of the use of topic tree handlers in the control client:

- To handle some subscription requests from other clients
- To receive messages sent from other clients through topics
- To receive notifications of clients attempting to use topics that do not exist
- To handle requests from clients for the state of delegated topics.

All topic tree handlers have an `onActive` callback that is used to pass a registered handler reference, which the application can use to close the registration when required. There is also an `onClose` callback, which is used to notify that the registration has been closed.

Content

Content is the generic term for data that is transmitted between sessions and the Diffusion server. Essentially, from the Diffusion point of view content is simply bytes and has no format imposed upon it by Diffusion.

Clients subscribe to topics receive the topic state and any updates as content. This content can be formatted according to the topic. The API provides readers to simplify the interpretation of certain types of content.

Clients that send and receive messages through topics can also represent the data body part of the message as content, but can also add optional headers if required.

A control client can use content to initialize or update the state of topics at the server.

Even though content is simply treated as bytes there are aids to formatting content within the API. The content factory provides builders and convenience methods available for creating certain types of content, such as simple string content or Diffusion record-based content.

When a control client uses content to update topics, it is important that the content is formatted in the way expected by the topic for updates.

Metadata in the Unified API

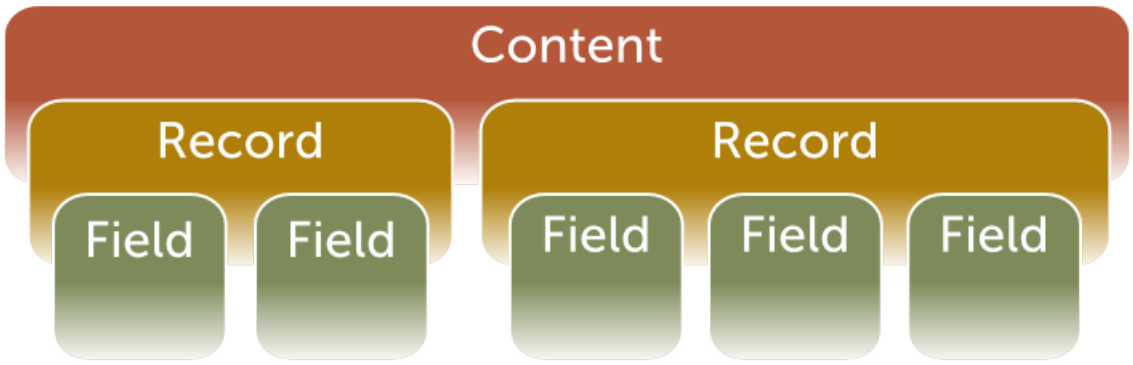
Metadata is available in the Unified API. It defines how the bytes in your content are formatted.

Content contains byte data. This byte data can be formatted in whatever way your application requires. For example, the data can structured using Diffusion record format.

You can use the provided Diffusion APIs to format the byte data as record-based data by defining a metadata structure that describes the data format.

Metadata structure

The metadata structure is made up of nested records and fields. The outer container is a the *content*. This contains zero, one, or many *records*. Each record can contain one or many *fields*.



Fields and records are identified by a name. Every record must have a name that is unique within the content. Every field must have a name that is unique within the enclosing record.

Every field or record defined in the metadata structure can represent one or more possible occurrences of that field or record in the byte data. The number of possible occurrences of a record or field is described by its multiplicity.

The order in which records and fields are defined within their enclosing container defines the order that they appear in byte data.

Field metadata

A metadata field defines an elementary data item within a record.

Every field has the following properties:

- Data type
- Multiplicity
- Default value

Data type

The data type of a field defines its actual representation within the byte data. The following table describes the data types that are available.

Table 69: Data types for metadata fields

Data type	Description	Default
String	A character string.	Zero-length string
Integer string	An integer represented in the content as a character string. If a field is defined as this type, it can only contain numeric digits with an optional leading sign. Fields of this type cannot be empty.	0

Data type	Description	Default
Decimal string	A decimal number represented in the content as a character string. Decimal fields have the number of places to the right of the decimal point defined by the scale, the default being 2. Such values can be parsed from a character string with any number of digits to the right of the decimal point. Half-up rounding is applied to achieve the target scale. Output of the field is rendered with the specified scale. Fields of this type cannot be empty. For comparison purposes the scale is ignored: a value of 1.50 is the same as 1.5.	0.00 (depending on scale)
Custom string	This is a special type where the behavior is delegated to a user-written custom field handler. This type is available in all topic data types.	-

Multiplicity

The multiplicity of a metadata field or record defines the number of times the corresponding byte data can occur within the enclosing record or content.

Multiplicity is defined in terms of the minimum and maximum number of occurrences. Some byte data representations support variable numbers of records and field, whereas others (such a record data) only support fixed number of records and fields (where minimum=maximum) except in the last position.

Fixed multiplicity is defined by a single number. For example, a multiplicity of 5 on a field indicates that there must be exactly five occurrences of the field within its enclosing record.

Variable multiplicity is defined by a minimum value and a maximum value and is represented with the notation *n*..*n*. For example, multiplicity of 1..5 on a field specifies that there can be between one and five occurrences of the field within its enclosing record.

A special maximum value of -1 is used to represent no maximum. For example, a multiplicity of 1..-1 on a field specifies there can be any number of occurrences of the field, but there must be at least one.

Optional nodes are defined by a minimum value of 0. For example, a multiplicity of 0..1 on a field specifies that there can be zero of one occurrences of the field within its enclosing record. A fixed multiplicity of 0 is not allowed.

Default value

You can specify a default value for a field. If you do not specify a default value, the default value for the data type is used. When content is created using metadata, default initialization applies the default values specified for each field.

Creating a metadata definition for a record topic

You can use the Java Unified API to specify the metadata structure that describes the byte data content of a message.

About this task

Control clients define the metadata structure for messages. This metadata structure can be used when defining a topic. All messages placed on the topic must conform to the metadata structure.

Note: Where there notation `c.p.d` is used in class or package names, it indicates `com.pushtechnology.diffusion`.

Procedure

1. Define the metadata structure.
   a) Import `c.p.d.client.Diffusion` and the following classes from the `c.p.d.client.content.metadata` package:
      - `MetadataFactory`
      - `MContent`
      - `MRecord`
      - `MField`
      - `MString`
      - `MIntegerString`
      - `MDecimalString`
      - `MCustomString`

   b) Use the `Diffusion.metadata` method to get a `MetadataFactory`.

```
private final MetadataFactory factory = Diffusion.metadata();
```

   c) Use the methods on the `MetadataFactory` to specify the content, record, and field definitions that make up the metadata structure.

      For example, the following code uses a content builder to create content metadata with a single record type that can occur zero to *n* times.

```
public MContent createContentRepeating() {
    return
        factory.contentBuilder("Content").
            add(
                factory.record(
                    "Rec1",
                    factory.string("A"),
                    factory.string("B")),
                0,
                -1).
            build();

}
```

      For more information, see Java Unified API documentation.
2. Create a record topic and apply the metadata definition to it.
   a) Import the TopicControl feature, `Session` class, and `RecordTopicDetails` class.

```
import
 com.pushtechnology.diffusion.client.features.control.topics.TopicControl;
import com.pushtechnology.diffusion.client.session.Session;
import
 com.pushtechnology.diffusion.client.topics.details.RecordTopicDetails;
```

   b) Create a `Session` instance and use it to get the TopicControl feature.

```
final Session session = Diffusion.sessions().open("dpt://
localhost:8081");
final TopicControl tc = session.feature(TopicControl.class);
```

   c) Get a topic builder for a record topic from the TopicControl feature.

```
final RecordTopicDetails.Builder builder =
            tc.newDetailsBuilder(RecordTopicDetails.Builder.class);
```

d) Use the `metadata` method of the topic builder to create the topic definition.

```
tc.addTopic(
            TOPIC_NAME,
            builder.metadata(metadata).build(),
            new TopicControl.AddCallback.Default() {
                @Override
                public void onTopicAdded(String topic) {
                    theTopic = topic;
                }

            });
```

Example: A control client that creates a metadata definition and uses it when creating a topic.

```
package com.example.metadata;

import com.pushtechnology.diffusion.client.Diffusion;
import
 com.pushtechnology.diffusion.client.content.metadata.MContent;
import
 com.pushtechnology.diffusion.client.content.metadata.MDecimalString;
import
 com.pushtechnology.diffusion.client.content.metadata.MField;
import
 com.pushtechnology.diffusion.client.content.metadata.MRecord;
import
 com.pushtechnology.diffusion.client.content.metadata.MetadataFactory;
import
 com.pushtechnology.diffusion.client.topics.details.TopicType;
import
 com.pushtechnology.diffusion.client.features.control.topics.TopicControl;
import com.pushtechnology.diffusion.client.session.Session;
import
 com.pushtechnology.diffusion.client.topics.details.RecordTopicDetails;
import
 com.pushtechnology.diffusion.client.types.UpdateOptions;


/**
 * An example of a control client creating metadata definition
 and using it when creating a
 * topic definition.
 */
public class ControlClient {

    private final MetadataFactory factory =
 Diffusion.metadata();

    /**
     * Creates control client instance.
     */
    public ControlClient() {

        // Create the Session
        final Session session = Diffusion.sessions()
                .open("dpt://localhost:8081");

        session.start();

        // Add the TopicControl feature
        final TopicControl tc =
 session.feature(TopicControl.class);
```

```
        // Create metadata for the topic
        final MContent metadata = defineMetadata();

        // Get a topic builder
        final RecordTopicDetails.Builder builder =

tc.newDetailsBuilder(RecordTopicDetails.Builder.class);

        // Create the topic with metadata
        tc.addTopic(
            TOPIC_NAME,
            builder.metadata(metadata).build(),
            new TopicControl.AddCallback.Default() {
                @Override
                public void onTopicAdded(String topic) {
                    theTopic = topic;
                }

            });

    }


    /**
     * A simple example of creating content metadata with two
records.
     *
     * @return content metadata
     */
    public MContent defineMetadata() {
        return factory.content(
            "Content",
            createNameAndAddressRecord(),
            createMultipleRateCurrencyRecord("Exchange Rates",
5));
    }

    /**
     * Creates a simple name and address record with fixed
name single
     * multiplicity fields.
     *
     * @return record definition.
     */
    public MRecord createNameAndAddressRecord() {
        return factory.record(
            "NameAndAddress",
            factory.string("FirstName"),
            factory.string("Surname"),
            factory.string("HouseNumber"),
            factory.string("Street"),
            factory.string("Town"),
            factory.string("State"),
            factory.string("PostCode"));
    }

    /**
     * This creates a record with two fields, a string called
"Currency" and a
     * decimal string called "Rate" with a default value of
1.00 which repeats a
     * specified number of times.
     *
     * @param name the record name
     * @param occurs the number of occurrences of the "Rate"
field
     * @return the metadata record
```

```
     */
    public MRecord createMultipleRateCurrencyRecord(String
 name, int occurs) {
        return factory.recordBuilder(name).
            add(factory.string("Currency")).
            add(factory.decimal("Rate", "1.00"), occurs).
            build();
    }

}
```

Using a custom field handler

You can use the Java Classic API to implement a class that handles data in custom fields. This class is located on the server and can be invoked by the Unified API.

About this task

> **Note:** Where there notation `c.p.d` is used in class or package names, it indicates `com.pushtechnology.diffusion`.

Custom field handlers implement the `c.p.d.api.data.metadata.CustomFieldHandler` interface.

Procedure

1. Create your custom field handler.

```
package com.example.custom

import
 com.pushtechnology.diffusion.api.data.metadata.CustomFieldHandler

public class DoubleFieldHandler implements CustomFieldHandler {

 public Object getInitialDefaultValue() {
  return new Double(0.0);
 }
 public Object parse(Object object) throws APIException {
  if (object==null) {
   return new Double(0.0);
  }

  try {
   return (Double.parseDouble(object.toString()));
  }
  catch (Throwable ex) {
   throw new APIException(
   "Unable to parse "+object+" as double value",
   ex);
  }
 }

 public boolean areEqual(Object source,Object target) {
  return source.equals(target);
 }
}
```

   a) Import `c.p.d.api.data.metadata.CustomFieldHandler`.
   b) Implement `CustomFieldHandler`.
   c) Implement the following methods:

      - `getInitialDefaultValue`, which returns the default value for the custom field
      - `parse`, which parses any object into the data type used by the custom field

- `areEqual`, which compares two objects of the data type used by the custom field or equality

2. Put your compiled custom field handler class in *`diffusion_installation`*`/ext` directory and restart the Diffusion server.
3. Use the custom field handler when defining metadata for a custom string field from your control client.

```
package com.example.metadata;

import com.pushtechnology.diffusion.client.Diffusion;
import com.pushtechnology.diffusion.client.content.metadata.MContent;
import
 com.pushtechnology.diffusion.client.content.metadata.MCustomString;
import
 com.pushtechnology.diffusion.client.content.metadata.MCustomString.Builder;
import com.pushtechnology.diffusion.client.content.metadata.MField;
import com.pushtechnology.diffusion.client.content.metadata.MRecord;
import
 com.pushtechnology.diffusion.client.content.metadata.MetadataFactory;

public class ControlClient {

    private final MetadataFactory factory = Diffusion.metadata();

    /**
     * Create control client instance.
     */
    public ControlClient() {

    }

    /**
     * Example of using a custom field builder.
     * This method creates a custom string that uses the
 DoubleFieldHandler class
     * and contains the default value.
     */
    public MCustomString createCustom(String name) {
        return factory.customBuilder(name,
 "com.example.custom.DoubleFieldHandler").build();
    }

    /**
     * A simple example of creating content metadata with two records.
     * The records contain custom string fields.
     *
     * @return content metadata
     */
    public MContent createContent() {
        return factory.content(
            "Content",
            factory.record(
                "Rec1",
                createCustom("Field1"),
                createCustom("Field2")),
            factory.record(
                "Rec2",
                createCustom("Field3"));
    }

}
```

a) Import the following classes and interfaces:

- `c.p.d.client.content.metadata.MetadataFactory;`

- `c.p.d.client.content.metadata.MCustomString`
- `c.p.d.client.content.metadata.MCustomString.Builder`

b) Get and instance of `MetadataFactory`.

c) Use the `customBuilder` method on the `MetadataFactory` to get a custom string builder. Pass in the name of the custom field and the fully qualified name of the custom field handler class you developed.

For example:

```
factory.customBuilder(name, "com.example.custom.DoubleFieldHandler")
```

d) Use the `build` method on the `MCustomString.Builder` to get the `MCustomString`. You can use this metadata custom field definition when specifying your metadata structure.

Topic details

Topic details is the generic term used for a description of a topic.

There are the following different levels of topic detail:

BASIC

Provides the most basic detail about a topic, such as its type.

SCHEMA

Provides the basic detail plus information about the format of the data maintained by the topic. For topic types that have their content layout described by metadata, this is where the metadata is defined. Many topic types (such as functional topics) do not have schema definitions.

FULL

Provides the schema level of detail plus attributes of the topic. A standard client rarely requires this level of detail. It is necessary only when creating topics using a control client.

When a standard client subscribes to any topic for the first time, it receives a notification of the basic topic details of that topic. This is guaranteed to arrive before the first update is received for that topic.

Any client can also explicitly request any level of detail for any topic.

Control clients use full level topic details to describe a topic when creating it. Builders (and convenience methods) are available for creating details relating to all the different topic types.

When a control client wants to create many different topics with the same description, we recommended that a single instance of a topic details object is used for every request. This is because there are caching optimizations at the server that mean that the same description is not transmitted to the server every time.

Topic types

The topic type defines what a topic is used for and how it behaves. Some topic types can have schema information and some can have attributes over and above the standard attributes available to all topic types.

The following topic types exist:

STATELESS

A topic that has no data held at the server and no particular function. It is generally used as an organizational node within the topic tree, but can also be used for sending messages.

> **Note:** When a topic is created with a hierarchic name, if the intermediate nodes in the hierarchy do not already exist they are automatically created as stateless topics.

DELEGATED

A topic that has no data held at the server. Its state is delegated to a control client. When a client subscribes or fetches, a control client that is registered to provide the state is invoked to retrieve the state. When a control client updates a delegated topic nothing is stored at the server and the update is simply fanned out to all subscribed clients.

SINGLE_VALUE

A topic that maintains state at the server as a single string value. The value can optionally be constrained in certain ways, for example, to hold an integer or decimal number. The nature of the single value is described using field metadata in the schema.

RECORD

A topic that maintains state at the server in Diffusion record format, which is effectively strings separated by field or record delimiters or both. This allows multiple fields to be maintained in the same topic and deltas of change are calculated at the server such that only those fields that have changed since the least update are sent out to the subscribed clients. The layout of the data is described using content metadata in the schema.

PROTOCOL_BUFFER

A topic that maintains state at the server in Google protocol buffers format. It benefits from the same delta processing capability at the server as a record topic. The schema defines a compiled proto class which must exist at the server and the name of a message definition within the class that defines the topic data layout. Google protocol buffer definitions cans be used to generate and interpret such content.

CUSTOM

A topic that has its state maintained server side by a user written Diffusion class. In many ways this makes it like a delegated topic except the state is delegated to an instance of the class and is all handled server side. Updates from a control client are also passed to the user handler for processing. The user handler can hold the topic state internally or elsewhere.

SLAVE

A special type of topic that has no state of its own but is a reference to the state of another topic. It is effectively a link to that master topic. Updates to the master are fanned out to subscribers of the slave and updates to the slave would update the master. The master must be one of the types of topic that maintains topic data.

SERVICE

A special type of topic that implements a request/response type service. The service is implemented as a user-written server-side Diffusion class.

PAGED_STRING

This is a special type of topic that maintains server-side state as a number of lines of string data. Clients using the Classic API can page through this data and have a current page that is updated if the state of that page changes. Paged topics can be created and updated using the Unified API.

PAGED_RECORD

A topic that is the same as a PAGED_STRING topic except each line of the data is defined in Diffusion record format. The schema defines the record metadata that defines the lines. Paged topics can be created and updated using the Unified API.

TOPIC_NOTIFY

A special type of topic that can be used by clients to receive notifications whenever topics are created or removed. Currently, though such a topic can be created using the Unified API, only clients implemented using the Classic API can use the topic.

ROUTING

A special type of topic, which can map to a different real topic for every client that subscribes to it. In this way, different clients can see different values for what is effectively the same topic from the client point of view. When a client subscribes to a routing topic, the request is either passed to a control client registered as a routing subscription handler for the topic or handled by a server-side routing handler.

CHILD_LIST

A special type of topic that maintains a list of its children and notifies subscribed clients when a child is added or removed. Though this type of topic can be created using the Unified API, it can be used only by Classic API clients.

Topic attributes

The attributes of a topic can be supplied when it is created, but in most cases the default values for the attributes are what is required.

There are a common set of attributes that apply to all topic types but certain types have additional attributes, some of which are mandatory. The API specifications of the relevant topic details builders indicate what is required.

The following topic attributes are common to all topic types:

Auto subscribe

Indicates whether clients that pre-emptively subscribe to the topic or any of its descendants automatically become subscribed to the topic when it is created. For topics created using the Unified API this is the default behavior, but it can be disabled. If this behavior is disabled, subscriptions must be performed manually after creating topics.

Tidy on unsubscribe

By default, if a client unsubscribes from a topic, it continues to receive any updates to that topic that were queued for it before it unsubscribed. By choosing this option, when a client unsubscribes from the topic any updates queued for that topic are removed from the client's queue.

Reference

You can associate a string reference with a topic which is used in logging and monitoring to make the topic easier to identify.

Properties

An optional map of properties can be supplied in order to specify topic attributes not supported by the Unified API. This capability is present for backwards compatibility. The key values of the properties are the same as

the names in the `TopicProperty` enum in the Java Classic API and the values are string representations of the property value. This cannot be used to set properties that are represented in the Unified API.

Supported client platforms

The Unified API is available for multiple client platforms. These client platforms have a supported version and support connection across a number of transport protocols.

Table 70: Tier 1 supported client platforms

Platform	Supported version	Supported transport protocols
Java	7, 8	• DPT, DPTS • WS, WSS • HTTP, HTTPS (Full duplex)
.NET	3.5	• DPT, DPTS • WS, WSS • HTTP (Full duplex)
C	-	• DPT

Java

The Java API comprises a number of classes subordinate to the `com.pushtechnology.diffusion.client` package.

The Java API is available in the following locations:

- The `diffusion-client-x.x.x.jar` file in the `clients/java` directory of your Diffusion installation. Where *x.x.x* is the Diffusion version, for example 5.1.0. This jar also includes the client libraries required to run a Diffusion client.
- The `diffusion-api-x.x.x.jar` file on Maven. Where *x.x.x* is the Diffusion version, for example 5.1.0. This jar includes only the interfaces required to develop your client. You can access this file by declaring the following dependency:

```
<dependency>
    <groupId>com.pushtechnology.diffusion</groupId>
    <artifactId>diffusion-api</artifactId>
    <version>x.x.x</version>
</dependency>
```

Full API documentation is issued with the product and this guide does not go into detail about how to use the interface. For more information, see Java Unified API documentation.

The following table describes the features and their interfaces. All interface names are relative to the `com.pushtechnology.diffusion.client.features` package.

Table 71: Java interfaces

Feature	Description	Relevant interfaces
AuthenticationControl	Use the AuthenticationControl feature to enable a	`control.clients.AuthenticationControl`

Feature	Description	Relevant interfaces
	control client session to authenticate other clients.	
ClientControl	Use the ClientControl feature to enable a control client session to receive notifications about other clients and to manage other clients.	`control.clients.ClientControl`
Messaging	Use the Messaging feature to enable a client session to send messages to a topic.	`Messaging`
MessagingControl	Use the MessagingControl feature to enable a control client to send messages to specific client sessions and receive messages sent by clients to topics.	`control.topics.MessagingControl`
Pings	Use the Pings feature to enable a client session to ping the server and verify the connection between client and server.	`Pings`
Security	Use the Security feature to enable a client to change the principal associated with a session.	`Security`
SubscriptionControl	Use the SubscriptionControl feature to enable a control client to subscribe other clients to topics and handle routing topic subscription requests.	`control.topics.SubscriptionControl`
TopicControl	Use the TopicControl feature to enable a control client session to create and manage topics and to provide state for delegated topics.	`control.topics.TopicControl`
Topics	Use the Topics feature to enable a client session to subscribe to topics and receive streaming update to	`Topics`

Feature	Description	Relevant interfaces
	those topics or to fetch the current state of topics.	
TopicUpdateControl	Use the TopicUpdateControl feature to enable a control client session to update topics.	`control.topics.TopicUpdateControl`

Related Links

Using Maven to build Java Diffusion applications on page 600
Apache Maven is a popular Java build tool and is well supported by Java IDEs. You can use Apache Maven to build your Diffusion applications.

Building client applications with Maven on page 600
You can build and run Diffusion Java client applications without installing the Diffusion product. The Diffusion client JAR is all you need.

Getting started with the Java API

The example demonstrates an empty Java client that you can base your clients on.

About this task

Procedure

1. Include the API jar file on the build path of your Java client. You can use one of the following methods:
   - You can use Maven to declare the dependency. For example:

     ```
     <dependency>
         <groupId>com.pushtechnology.diffusion</groupId>
         <artifactId>diffusion-api</artifactId>
         <version>x.x.x</version>
     </dependency>
     ```

   - You can include the `diffusion-client-x.x.x.jar` file that is located in the `clients/java` directory of your Diffusion server installation.

   Where *x.x.x* is the Diffusion version, for example 5.1.0.
2. Create a client class that has a `main` method.

```
import com.pushtechnology.diffusion.client.Diffusion;
import com.pushtechnology.diffusion.client.session.Session;
import com.pushtechnology.diffusion.client.session.SessionFactory;

public class Client {
    public static void main(String ... arguments) throws Exception {

        SessionFactory sf1 = Diffusion.sessions();

        sf1.connectionTimeout(1200)
           .inputBufferSize(100000);

        Session session = sf1.open("dpt://localhost:8081");

        // Set up any features that your session uses

        session.start();

        // Use the session
```

```
            session.close();
        }
    }
```

a) Import the `Diffusion`, `session.Session`, and `session.SessionFactory` classes.
b) In the `main` method, use the `Diffusion` object to get a `SessionFactory`.
c) Use the methods on the `SessionFactory` to configure the session.
d) Use the `open` method on the `SessionFactory` to get a `Session` instance.
e) Use the features provided by the Java API to set up the listeners, handlers and methods that perform your client's actions.

   For more information about the available features, see Features on page 266.
f) Start the session to connect the client to the Diffusion server.
g) Use the session to perform your client actions.
h) Close the session to disconnect the client from the server.

Related Links

Using Maven to build Java Diffusion applications on page 600
Apache Maven is a popular Java build tool and is well supported by Java IDEs. You can use Apache Maven to build your Diffusion applications.

Building client applications with Maven on page 600
You can build and run Diffusion Java client applications without installing the Diffusion product. The Diffusion client JAR is all you need.

.NET

The .NET API comprises a number of assemblies subordinate to the `PushTechnology.ClientInterface.Client` assembly.

API documentation is issued with the product and this guide does not go into detail about how to use the interface. For more information, see .NET Unified API documentation.

The following table describes which assemblies contain the major interface features. All interface names are relative to the `PushTechnology.ClientInterface.Client.Features` package.

Table 72: .NET interfaces

Feature	Description	Relevant interfaces
AuthenticationControl	Use the AuthenticationControl feature to enable a control client session to authenticate other clients.	`Control.Clients.IAuthenticationControl`
ClientControl	Use the ClientControl feature to enable a control client session to receive notifications about other clients and to manage other clients.	`Control.Clients.IClientControl`
Messaging	Use the Messaging feature to enable a client session to send messages to a topic.	`IMessaging`

Feature	Description	Relevant interfaces
MessagingControl	Use the MessagingControl feature to enable a control client to send messages to specific client sessions and receive messages sent by clients to topics.	`Control.Clients.IMessagingControl`
Pings	Use the Pings feature to enable a client session to ping the server and verify the connection between client and server.	`IPings`
Security	Use the Security feature to enable a client to change the principal associated with a session.	`ISecurity`
SubscriptionControl	Use the SubscriptionControl feature to enable a control client to subscribe other clients to topics and handle routing topic subscription requests.	`Control.Topics.ISubscriptionControl`
TopicControl	Use the TopicControl feature to enable a control client session to create and manage topics and to provide state for delegated topics.	`Control.Topics.ITopicControl`
Topics	Use the Topics feature to enable a client session to subscribe to topics and receive streaming update to those topics or to fetch the current state of topics.	`ITopics`
TopicUpdateControl	Use the TopicUpdateControl feature to enable a control client session to update topics.	`Control.Topics.ITopicUpdateControl`

Getting started with the .NET API

The example demonstrates an empty .NET client that you can base your clients on.

Procedure

1. Create a .NET project that references the following DLL files located in the `clients/dotnet` directory of your .NET installation:

   `PushTechnology.ClientInterface.dll`
   The assembly contains the interfaces classes and interfaces that you use when creating a control client.

   `PushTechnology.DiffusionBaseUtils.dll`, **`PushTechnology.DiffusionCore.dll`**, **`PushTechnology.DiffusionExternalClient.dll`**
   These assemblies are used to build your control client. Do not use any of the classes or interfaces they contain when writing your control client.

2. In your project, create a C# file that contains the `Main` method.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Diagnostics;
using PushTechnology.ClientInterface.Client.Session;
using PushTechnology.ClientInterface.Client.Features;
using PushTechnology.ClientInterface.Client.Factories;

namespace PushTechnology.Examples
{
    class Client
    {
        private static ISessionFactory sessionFactory;
        private static ISession session;

        static void GetStarted()
        {
            sessionFactory = Diffusion.Sessions
                .ConnectionTimeout(1200)
                .InputBufferSize(100000);
            session = sessionFactory.Open("dpt://localhost:8081");

            // Set up any features that your session uses
            session.Start();

            // Use the session
            session.Close();

        }
    }
}
```

   a) Use `PushTechnology.ClientInterface.Client.Session` and `PushTechnology.ClientInterface.Client.Features`.
   b) Get an `ISessionFactory` instance using the `Diffusion.Sessions()` method.
   c) Use the methods on the session factory to configure the session.
   d) Use the `Open` method on the session factory to get an `ISession` instance.
   e) Use the features provided by the .NET API to set up the listeners, handlers and methods that perform your client's actions.

      You can get instances of the features by using the `Get`*`Name`*`Feature` methods of the `Session`, where *`Name`* is the name of the feature.

For more information about the available features, see Features on page 266.

f) Start the session to connect the client to the Diffusion server.
g) Use the session to perform your client actions.
h) Close the session to disconnect the client from the server.

C

The C API is provided in the `diffusion-c-x.x.x.zip` file, where `x.x.x` is the version number, for example 5.1.0. This file is located in the `clients/c` directory of your Diffusion installation. The zip file contains a static library, a dynamic library, and a number of header files.

A C client created with the Unified API can connect to Diffusion only using the DPT protocol.

The libraries are compiled on 64-bit Linux and are supported on Red Hat 6.5 and CentOS 6.5. If you require libraries compiled on a different platform, contact support@pushtechnology.com.

API documentation is issued with the product and this guide does not go into detail about how to use the interface. For more information, see C Unified API documentation.

The following table describes what functions are provided by the C API.

Table 73: C functions

Feature	Description
AuthenticationControl	Use the AuthenticationControl feature to enable a control client session to authenticate other clients.
Messaging	Use the Messaging feature to enable a client session to send messages to a topic.
MessagingControl	Use the MessagingControl feature to enable a control client to send messages to specific client sessions and receive messages sent by clients to topics.
Pings	Use the Pings feature to enable a client session to ping the server and verify the connection between client and server.
Security	Use the Security feature to enable a client to change the principal associated with a session.
Partial support: TopicControl	Use the TopicControl feature to enable a control client session to create and manage topics and to provide state for delegated topics.
Topics	Use the Topics feature to enable a client session to subscribe to topics and receive streaming update to those topics or to fetch the current state of topics.
Partial support: TopicUpdateControl	Use the TopicUpdateControl feature to enable a control client session to update topics.

Getting started with the C API

The example demonstrates an empty C client that you can base your clients on.

Before you begin

The C client libraries rely on the following libraries:

- Apache Portable Runtime (APR) version 1.5 with APR-util library

For more information, see http://apr.apache.org

- Perl Compatible Regular Expressions (PCRE) library, version 8.35

  For more information, see http://pcre.org

Ensure that these libraries are available on your development system. You can download them through your operating system's package manager.

Procedure

1. Create a C file that contains the client `main` method. For example, `example.c`.

```
#include <stdio.h>
#include <unistd.h>

#include "diffusion.h"
#include "args.h"

ARG_OPTS_T arg_opts[] = {
        ARG_OPTS_HELP,
        {'u', "url", "Diffusion server URL", ARG_OPTIONAL,
 ARG_HAS_VALUE, "dpt://localhost:8081"},
        END_OF_ARG_OPTS
};

int
main(int argc, char **argv)
{
        // Standard command line parsing.
        HASH_T *options = parse_cmdline(argc, argv, arg_opts);
        if(options == NULL || hash_get(options, "help") != NULL) {
                show_usage(argc, argv, arg_opts);
                return 1;
        }

        char *url = hash_get(options, "url");

        // A SESSION_LISTENER_T holds callbacks to inform the client
        // about changes to the state. Used here for informational
        // purposes only.
        SESSION_LISTENER_T session_listener;
        session_listener.on_state_changed = &on_session_state_changed;
        session_listener.on_state_error = &on_session_state_error;

        // Create a session with the Diffusion server.
        DIFFUSION_ERROR_T error;
        SESSION_T *session = NULL;
        session = session_create(url, NULL, NULL, &session_listener,
 NULL, &error);
        if(session == NULL) {
                fprintf(stderr, "TEST: Failed to create session\n");
                fprintf(stderr, "ERR : %s\n", error.message);
                return 1;
        }

 // Set up the features you want to use with your client session.

        // Start the session.
        if(! session_start(session, &error)) {
                fprintf(stderr, "ERR: Failed to start session: %s\n",
 error.message);
                return 1;
        }

        // Use the session and features.
        // ...
```

```
        // Close the session with the Diffusion server
        session_close(session, &error);

        return 0;
}
```

a) Include the `diffusion.h` header file.

   This file pulls in the other required header files.

b) In the `main` method, use the `session_create` function to create and configure the session.

   The Diffusion server connection URL is the only mandatory parameter for this function. All other parameters can take a `NULL` value.

c) Do any setup on the session as required. For example, subscribe to topics.

d) Use the `session_start` function to start the session.

e) Use the session to perform your client actions.

   All functions that use the client features take the `session` as a parameter in addition to feature-specific arguments and handlers.

f) Use the `session_close` function to disconnect the client from the server.

2. Create a makefile to build your client.

```
CC  = gcc
AR  = ar
CFLAGS  = -c -g -Wall -I../public/include -I/usr/include/apr-1
LDFLAGS  = -L../lib -lapr-1 -laprutil-1 -lpcre -ldiffusion
ARFLAGS  =
SOURCES  = example.c
OBJECTS  = $(SOURCES:.c=.o)
TARGETS  = example

all:  $(SOURCES) $(TARGETS)

.c.o:
  $(CC) $(CFLAGS) $< -o $@

clean:
  rm -f $(OBJECTS) $(TARGETS) core a.out
```

a) Link against one of the Diffusion libraries:

- `libdiffusion.so` for dynamic binding
- `libdiffusion.a` for static binding

b) Link against all of the following libraries:

- apr
- apr-util
- pcre

Features

The Unified API is organized into features that make up conceptual groupings of capabilities.

The delineation between features enables the clients to start only those services that it uses.

Each feature contains interfaces that you can implement to use the capabilities of the feature, default implementations of these interfaces that perform basic logging, and enums that list allowed values that are used or returned by methods in the feature.

The Unified API provides features in a variety of languages.

Table 74: Matrix of supported features by language

Feature			
AuthenticationControl	YES	YES	YES
ClientControl	YES	YES	NO
Messaging	YES	YES	YES
MessagingControl	YES	YES	PARTIAL Does not support sending messages to a specific client.
Pings	YES	YES	YES
Security	YES	YES	YES
SubscriptionControl	YES	YES	NO
TopicControl	YES	YES	PARTIAL Does not support paged topics
Topics	YES	YES	YES
TopicUpdateControl	YES	YES	PARTIAL Does not support paged topics

AuthenticationControl

Use the AuthenticationControl feature to enable a control client session to authenticate other clients.

The AuthenticationControl feature is available in the following APIs:

Feature	Java	.NET	C
AuthenticationControl	YES	YES	YES

The AuthenticationControl feature contains the following classes and interfaces:

`ControlAuthenticationHandler`
: Implement this interface to handle authentication requests from other clients.

`CompositeControlAuthenticationHandler`
: Extend this class to delegate authentication requests to one or more control authentication handlers.

The following examples show using the Unified API to register a control authentication handler with the server. The examples also include simple or empty authentication handler.

The name by which the control authentication handler is registered, example-handler, must be configured in the `Server.xml` configuration file of the Diffusion server for the control authentication handler to be called to handle authentication requests.

Java

```
package com.pushtechnology.diffusion.examples;
```

```
import java.nio.charset.Charset;
import java.util.EnumSet;

import com.pushtechnology.diffusion.client.Diffusion;
import com.pushtechnology.diffusion.client.details.SessionDetails;
import
 com.pushtechnology.diffusion.client.details.SessionDetails.DetailType;
import com.pushtechnology.diffusion.client.features.ServerHandler;
import
 com.pushtechnology.diffusion.client.features.control.clients.AuthenticationControl;
import
 com.pushtechnology.diffusion.client.features.control.clients.AuthenticationControl.C
import com.pushtechnology.diffusion.client.session.Session;
import com.pushtechnology.diffusion.client.types.Credentials;

/**
 * This demonstrates the use of a control client to authenticate client
 * connections.
 *
 * This uses the 'AuthenticationControl' feature.
 *
 * @author Push Technology Limited
 */
public class ControlClientIdentityChecks {

    private final Session theSession;

    /**
     * Constructor.
     */
    public ControlClientIdentityChecks() {

        theSession = Diffusion.sessions().open("dpt://localhost:8081");

        final AuthenticationControl authenticationControl =
            theSession.feature(AuthenticationControl.class);

        /**
         * Authentication handler
         */
        class Handler extends ServerHandler.Default
            implements ControlAuthenticationHandler {
            @Override
            public void authenticate(
                final String principal,
                final Credentials credentials,
                final SessionDetails sessionDetails,
                final Callback callback) {

                final byte[] passwordBytes =
                    "password".getBytes(Charset.forName("UTF-8"));

                if ("Admin".equals(principal) &&
                    credentials.getType() ==
 Credentials.Type.PLAIN_PASSWORD &&
                    credentials.toBytes().equals(passwordBytes)) {
                    callback.allow();
                }
                else {
                    callback.deny();
                }
            }
        }

        authenticationControl.setAuthenticationHandler(
            "example-handler",
            EnumSet.allOf(DetailType.class),
```

```
            new Handler());

        theSession.start();
    }

    /**
     * Close the session.
     */
    public void close() {
        theSession.close();
    }
}
```

.NET

```
using System;
using System.Collections.Generic;
using PushTechnology.ClientInterface.Client.Session;
using PushTechnology.ClientInterface.Client.Features;
using PushTechnology.ClientInterface.Client.Types;
using PushTechnology.ClientInterface.Client.Features.Control.Clients;
using PushTechnology.ClientInterface.Client.Details;
using PushTechnology.ClientInterface.Client.Security.Authentication;
using PushTechnology.ClientInterface.Client.Factories;

namespace PushTechnology.Examples
{
    // A control client that uses the AuthenticationControl feature
    // to register an authentication handler that can authenticate other
    // clients.
    public class ControlClientHandlingAuthentication
    {
        private ISession session;
        private IAuthenticationControl authControlFeature;
        private IControlAuthenticationHandler authHandler;
        private List<DetailType> requestedDetails;

        public ControlClientHandlingAuthentication()
        {
        }

        public void Open()
        {
            session = Diffusion.Sessions
                .ConnectionTimeout(1200)
                .InputBufferSize(100000)
                .Open("dpt://localhost:8081");
            session.Start();

            // Use the session to get the feature
            authControlFeature =
 session.GetAuthenticationControlFeature();

            // Get an instance of an authentication handler.
            // Authentication handlers registered by a control client must
            // implement the IControlAuthenticationHandler interface
            authHandler = new MyCompositeControlAuthenticationHandler();

            // Specify the details that the authentication handler
            // requires from the server to authenticate a client
            requestedDetails = new List<DetailType>();
            requestedDetails.Add(DetailType.SUMMARY);

            //Register the authentication handler by name.
            // The name given to the authentication handler,
            // "demoHandler"
```

```
            authControlFeature.SetAuthenticationHandler("demoHandler",
 requestedDetails, authHandler);
        }

        public void Close()
        {
            session.Close();
        }
    }


    // A composite authentication handler that delgates to the
 ControlAuthenticationHandler.

    class MyCompositeControlAuthenticationHandler :
 CompositeControlAuthenticationHandler
    {
        public MyCompositeControlAuthenticationHandler() :
            base(new ControlAuthenticationHandler())
        {
        }
    }


    // An authentication handler that allows all connections.

    class ControlAuthenticationHandler : IControlAuthenticationHandler
    {
        public void Authenticate(string principal,
 ICredentials credentials, ISessionDetails sessionDetails,
 IAuthenticationHandlerCallback callback)
        {
            // do the authentication
            callback.Allow();
        }

        public void OnActive(IRegisteredHandler registeredHandler)
        {
            // registered and active, make sure you open database
 connections (for example)
        }

        public void OnClose()
        {
            // not registered anymore, you can close the database
 connections (for example)
        }
    }
}
```

C

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include "diffusion.h"
#include "args.h"
#include "conversation.h"

struct user_credentials_s {
 const char *username;
 const char *password;
};

static const struct user_credentials_s USERS[] = {
```

```
 { "fish", "chips" },
 { "ham", "eggs" },
 { NULL, NULL }
};

ARG_OPTS_T arg_opts[] = {
 ARG_OPTS_HELP,
 {'u', "url", "Diffusion server URL", ARG_OPTIONAL, ARG_HAS_VALUE, "dpt://
localhost:8081"},
 {'n', "name", "Name under which to register the authorisation handler",
 ARG_OPTIONAL, ARG_HAS_VALUE, "c-auth-handler"},
 {'p', "principal", "Principal (username) for the connection",
 ARG_OPTIONAL, ARG_HAS_VALUE, NULL},
 {'c', "credentials", "Credentials (password) for the connection",
 ARG_OPTIONAL, ARG_HAS_VALUE, NULL},
 END_OF_ARG_OPTS
};

/**
 * When the authentication service has been registered, this function will
 be
 * called.
 */
static int
on_registration(SESSION_T *session, const
 SVC_AUTHENTICATION_REGISTER_RESPONSE_T *response)
{
 printf("Registered authentication handler\n");
 return HANDLER_SUCCESS;
}

/**
 * When the authentication service has be deregistered, this function will
 be
 * called.
 */
static int
on_deregistration(SESSION_T *session, const
 SVC_AUTHENTICATION_DEREGISTER_RESPONSE_T *response)
{
 printf("Deregistered authentication handler\n");
 return HANDLER_SUCCESS;
}

/**
 * This is the function that is called when authentication has been
 delegated
 * from Diffusion.
 *
 * The response may return one of three values via the response parameter:
 * ALLOW:   The user is authenticated.
 * DENY:    The user is NOT authenticated.
 * ABSTAIN: Allow another handler to make the decision.
 *
 * The handler should return HANDLER_SUCCESS in all cases, unless an
 actual
 * error occurs during the authentication process (in which case,
 * HANDLER_FAILURE is appropriate).
 */
static int
on_authentication(SESSION_T *session,
 const SVC_AUTHENTICATION_REQUEST_T *request,
 SVC_AUTHENTICATION_RESPONSE_T *response)
{
 int auth_decided = 0;

 // No credentials, or not password type. We're not an authority for
```

```
 // this type of authentication so abstain in case some other registered
 // authentication handler can deal with the request.
 if(request->credentials == NULL) {
  printf("No credentials specified, abstaining\n");
  response->value = AUTHENTICATION_ABSTAIN;
  auth_decided = 1;
  return HANDLER_SUCCESS;
 }
 if(request->credentials->type != PLAIN_PASSWORD) {
  printf("Credentials are not PLAIN_PASSWORD, abstaining\n");
  response->value = AUTHENTICATION_ABSTAIN;
  auth_decided = 1;
  return HANDLER_SUCCESS;
 }

 printf("principal = %s\n", request->principal);
 printf("credentials = %*s\n",
  (int)request->credentials->data->len,
  request->credentials->data->data);

 if(request->principal == NULL || strlen(request->principal) == 0) {
  printf("Denying anonymous connection (no principal)\n");
  response->value = AUTHENTICATION_DENY; // Deny anon connections
  auth_decided = 1;
  return HANDLER_SUCCESS;
 }

 char *password = malloc(request->credentials->data->len + 1);
 memcpy(password, request->credentials->data->data, request->credentials-
>data->len);
 password[request->credentials->data->len] = '\0';

 int i = 0;
 while(USERS[i].username != NULL) {

  printf("Checking username %s vs %s\n", request->principal,
 USERS[i].username);
  printf("     and password %s vs %s\n", password, USERS[i].password);

  if(strcmp(USERS[i].username, request->principal) == 0 &&
   strcmp(USERS[i].password, password) == 0) {

   puts("Allowed");
   response->value = AUTHENTICATION_ALLOW;
   auth_decided = 1;
   break;
  }
  i++;

 }

 free(password);

 if(auth_decided == 0) {
  puts("Abstained");
  response->value = AUTHENTICATION_ABSTAIN;
  auth_decided = 1;
 }

 return HANDLER_SUCCESS;
}

int
main(int argc, char** argv)
{
 HASH_T *options = parse_cmdline(argc, argv, arg_opts);
 if (options == NULL || hash_get(options, "help") != NULL) {
```

```
 show_usage(argc, argv, arg_opts);
 return 1;
}

char *url = hash_get(options, "url");
char *name = hash_get(options, "name");
char *principal = hash_get(options, "principal");
char *credentials = hash_get(options, "credentials");

// Create a session with Diffusion.
puts("Creating session");
DIFFUSION_ERROR_T error;
SESSION_T *session = session_create(url,
 principal,
 credentials != NULL ? credentials_create_password(credentials) : NULL,
 NULL, NULL,
 &error);
if (session == NULL) {
 fprintf(stderr, "TEST: Failed to create session\n");
 fprintf(stderr, "ERR : %s\n", error.message);
 return 1;
}

// This declares the handlers that get called when the auth handler has
// been successfully registered with Diffusion.
AUTHENTICATION_REGISTRATION_HANDLERS_T *registration_handlers = calloc(1,
sizeof(AUTHENTICATION_REGISTRATION_HANDLERS_T));
registration_handlers->on_registration = on_registration;

// Provide a set (via a hash map containing keys and NULL values)
// to indicate what information about the connecting client that we'd
// like Diffusion to send us.
HASH_T *detail_set = hash_new(5);
char buf[2];
sprintf(buf, "%d", SESSION_DETAIL_SUMMARY);
hash_add(detail_set, strdup(buf), NULL);
sprintf(buf, "%d", SESSION_DETAIL_LOCATION);
hash_add(detail_set, strdup(buf), NULL);
sprintf(buf, "%d", SESSION_DETAIL_CONNECTOR_NAME);
hash_add(detail_set, strdup(buf), NULL);

// This is the handler that will get called when an authentication
// request is received.
AUTHENTICATION_HANDLERS_T *auth_handlers = calloc(1,
sizeof(AUTHENTICATION_HANDLERS_T));
auth_handlers->on_authentication = on_authentication;

// Register the authentication handler.
puts("Sending registration request");
SVC_AUTHENTICATION_REGISTER_REQUEST_T *reg_request =
 svc_authentication_register(session, name, registration_handlers,
detail_set, auth_handlers);

puts("Starting session");
// Start the session - begin receiving messages from Diffusion.
if (!session_start(session, &error)) {
 fprintf(stderr, "ERR: Failed to start session: %s\n", error.message);
 return 1;
}

sleep(10);

AUTHENTICATION_DEREGISTRATION_HANDLERS_T *deregistration_handlers =
calloc(1, sizeof(AUTHENTICATION_DEREGISTRATION_HANDLERS_T));
deregistration_handlers->on_deregistration = on_deregistration;

printf("Deregistering authentication handler\n");
```

```
 svc_authentication_deregister(session, reg_request,
 deregistration_handlers);

 // Never exit
 while (1) {
  sleep(10);
 }

 // Not called, but this is the way you would gracefully terminate the
 // connection with Diffusion.
 session_close(session, &error);

 return(EXIT_SUCCESS);
}
```

Related Links

Authentication handlers on page 418
You can implement authentication handlers that authenticate client connections to the Diffusion server or perform an action that requires authentication.

Control client on page 226
A Diffusion client application can take on the role of control client.

Authentication on page 415
You can implement and register handlers to authenticate clients when the clients try to perform operations that require authentication.

Authentication handlers on page 418
You can implement authentication handlers that authenticate client connections to the Diffusion server or perform an action that requires authentication.

Developing a control authentication handler on page 424
Implement the `ControlAuthenticationHandler` interface to create a control authentication handler.

Developing a composite control authentication handler on page 426
Extend the `CompositeControlAuthenticationHandler` class to combine the decisions from multiple control authentication handlers.

ClientControl

Use the ClientControl feature to enable a control client session to receive notifications about other clients and to manage other clients.

The ClientControl feature is available in the following APIs:

Feature	Java	.NET	C
ClientControl	YES	YES	NO

The ClientControl feature contains the following classes and interfaces:

`QueueEventHandler`
: Implement this interface to create a handler for client queue events. The Unified API provides a default implementation that logs client queue events.

`SessionDetailsCallback`
: Implement this interface to provide a callback that receives replies from client control methods. The Unified API provides a default implementation that logs the callback.

SessionDetailsContextCallback

Implement this interface to provide a callback that receives replies from client control methods. An object can be associated with this callback that provides the context of the call. The Unified API provides a default implementation that logs the callback.

SessionDetailsListener

Implement this interface to create a listener that receives notifications if client session details change. The Unified API provides a default implementation that logs client session details.

CloseReason

This enum lists the reasons that a server can provide for a client connection being closed.

The following examples use the Unified API to set queue thresholds and register listeners for queue events:

Java

```
package com.pushtechnology.diffusion.examples;

import static
 com.pushtechnology.diffusion.client.features.control.clients.MessageQueuePolicy.Thro

import com.pushtechnology.diffusion.client.Diffusion;
import
 com.pushtechnology.diffusion.client.features.control.clients.ClientControl;
import
 com.pushtechnology.diffusion.client.features.control.clients.ClientControl.ClientCal
import
 com.pushtechnology.diffusion.client.features.control.clients.ClientControl.QueueEver
import
 com.pushtechnology.diffusion.client.features.control.clients.MessageQueuePolicy;
import com.pushtechnology.diffusion.client.session.Session;
import com.pushtechnology.diffusion.client.session.SessionId;

/**
 * This demonstrates the use of a control client to apply both throttling
 and
 * conflation to clients. It throttles and conflates all clients that
 reach
 * their queue thresholds and remove when they go down again.
 *
 * This uses the 'ClientControl' feature.
 *
 * @author Push Technology Limited
 */
public class ControlClientConflateAndThrottle {

    private final Session theSession;
    private final ClientControl theClientControl;
    private final ClientCallback theClientCallback;

    /**
     * Constructor.
     */
    public ControlClientConflateAndThrottle() {

        // Create the session
        theSession = Diffusion.sessions().open("dpt://localhost:8081");

        // Create the ClientControl feature with a handler that sets queue
```

```
        // thresholds on new connecting clients and sets a listener for
 queue
        // events.
        theClientControl = theSession.feature(ClientControl.class);
        theClientCallback = new ClientCallback.Default();

        theClientControl.setQueueEventHandler(new MyThresholdHandler());

        theSession.start();
    }

    /**
     * Close the session.
     */
    public void close() {
        theSession.close();
    }

    private class MyThresholdHandler extends QueueEventHandler.Default {

        @Override
        public void onUpperThresholdCrossed(
            final SessionId client,
            final MessageQueuePolicy policy) {

            // The setThrottled method enables throttling and conflation.
            theClientControl.setThrottled(client, MESSAGE_INTERVAL, 10,
                theClientCallback);
        }

        @Override
        public void onLowerThresholdCrossed(
            final SessionId client,
            final MessageQueuePolicy policy) {

            // The setThrottled method enables throttling and conflation.
            theClientControl
                .setThrottled(client, MESSAGE_INTERVAL, 1000,
 theClientCallback);
        }
    }
}
```

.NET

```
using System;
using PushTechnology.ClientInterface.Client.Factories;
using PushTechnology.ClientInterface.Client.Session;
using PushTechnology.ClientInterface.Client.Features;
using PushTechnology.ClientInterface.Client.Enums;
using PushTechnology.ClientInterface.Client.Types;
using PushTechnology.ClientInterface.Client.Features.Control.Clients;
using PushTechnology.ClientInterface.Client.Details;
using PushTechnology.ClientInterface.Client.Security.Authentication;

namespace PushTechnology.Examples
{
    // A control client that applies client throttling to any client that
    // has a queue larger than the lower threashold. When the upper
    // threashold is reached more aggressive throttling is applied.
    public class ControlClientConflateAndThrottle
    {
        private ISession session;

        public ControlClientConflateAndThrottle()
        {
```

```
        }

        public void Open()
        {
            session = Diffusion.Sessions
                .ConnectionTimeout(1200)
                .InputBufferSize(100000)
                .Open("dpt://localhost:8081");
            session.Start();

            var clientControl = session.GetClientControlFeature();
            clientControl.SetQueueEventHandler(new
ThresholdHandler(clientControl));
        }

        public void Close()
        {
            session.Close();
        }

    }

    // A handler for client message queue events that applies throttling
to clients with large queues.
    public class ThresholdHandler : IQueueEventHandler
    {
        private IClientControl clientControl;
        private IClientCallback callback;

        public ThresholdHandler(IClientControl clientControl)
        {
            this.clientControl = clientControl;
            this.callback = new NoOpCallback();
        }

        public void OnUpperThresholdCrossed(SessionId client,
PushTechnology.ClientInterface.CommandServices.Commands.Control.Client.IMessageQueue
policy)
        {
            clientControl.SetThrottled(client,
ThrottlerType.MESSAGE_INTERVAL, 10, callback);
        }

        public void OnLowerThresholdCrossed(SessionId client,
PushTechnology.ClientInterface.CommandServices.Commands.Control.Client.IMessageQueue
policy)
        {
            clientControl.SetThrottled(client,
ThrottlerType.MESSAGE_INTERVAL, 1000, callback);
        }

        public void OnActive(IRegisteredHandler registeredHandler)
        {
            // Actions to take before queue notfications are received by
the control client
        }

        public void OnClose()
        {
            // Actions to take after the handler is closed
        }
    }

    // No-op client callback that ignores success or failure of the client
operations
    public class NoOpCallback : IClientCallback
    {
```

```
        public void OnComplete()
        {
        }

        public void OnDiscard()
        {
        }
    }
}
```

Related Links

Managing clients from a control client on page 237
A control client can use the ClientControl feature of the Unified API to manage other client sessions.

Control client on page 226
A Diffusion client application can take on the role of control client.

Messaging

Use the Messaging feature to enable a client session to send messages to a topic.

The Messaging feature is available in the following APIs:

Feature	Java	.NET	C
Messaging	YES	YES	YES

The Messaging feature contains the following classes and interfaces:

`SendCallback`

Implement this interface to provide a callback that receives success or failure notifications from send calls. The Unified API provides a default implementation that logs the callback.

`SendContextCallback`

Implement this interface to provide a callback that receives success or failure notifications from send calls. An object can be associated with this callback that provides the context of the call. The Unified API provides a default implementation that logs the callback.

`MessageStream`

Implement this interface to create a stream that receives notifications of messages sent through topics. For more information, see Topic basics on page 127.

DEPRECATED:
`Listener`

This capability is replaced by `MessageStream`.

Implement this interface to create a listener that receives notifications of messages sent through topics. For more information, see Topic basics on page 127.

The following examples use the Unified API to send a message to a topic:

Java

```
package com.pushtechnology.diffusion.examples;

import com.pushtechnology.diffusion.client.Diffusion;
import com.pushtechnology.diffusion.client.features.Messaging;
import
 com.pushtechnology.diffusion.client.features.Messaging.SendCallback;
```

```
import
 com.pushtechnology.diffusion.client.features.Messaging.SendContextCallback;
import com.pushtechnology.diffusion.client.session.Session;

/**
 * This is a simple example of a client that uses the 'Messaging' feature
 to
 * send messages to a topic.
 *
 * @author Push Technology Limited
 */
public final class ClientSendingMessages {

    private final Session theSession;
    private final Messaging theMessaging;
    private final SendCallback theCallback;
    private final SendContextCallback<String> theContextCallback;

    /**
     * Constructs a message sending application.
     *
     * @param callback an object to which callbacks from sends can be
 directed
     *
     * @param contextCallback an object to which callbacks from contextual
 sends
     *        can be directed
     */
    public ClientSendingMessages(
        SendCallback callback,
        SendContextCallback<String> contextCallback) {

        theCallback = callback;
        theContextCallback = contextCallback;

        // Create the session and get the Messaging feature
        theSession = Diffusion.sessions().open("dpt://localhost:8081");
        theMessaging = theSession.feature(Messaging.class);
        theSession.start();
    }

    /**
     * Sends a simple string message to a specified topic.
     *
     * There is no context with the message so callback is directed to
     * the no-context callback.
     *
     * @param topicPath the topic path
     * @param message the message to send
     */
    public void send(String topicPath, String message) {
        theMessaging.send(
            topicPath,
            Diffusion.content().newContent(message),
            theCallback);
    }

    /**
     * Sends a simple string message to a specified topic with context
 string.
     *
     * Callback is directed to the contextual callback with the string
     * provided.
     *
     * @param topicPath the topic path
     * @param message the message to send
     * @param context the context string to return with the callback
```

```
     */
    public void send(String topicPath, String message, String context) {
        theMessaging.send(
            topicPath,
            Diffusion.content().newContent(message),
            context,
            theContextCallback);
    }

    /**
     * Sends a string message to a specified topic with headers.
     *
     * There is no context with the message so callback is directed to
     * the no context callback.
     *
     * @param topicPath the topic path
     * @param message the message to send
     * @param headers the headers to send with the message
     */
    public void sendWithHeaders(
        String topicPath,
        String message,
        String... headers) {

        theMessaging.send(
            topicPath,
            Diffusion.content().newContent(message),
            theMessaging.sendOptionsBuilder().headers(
                Diffusion.content().newHeaders(headers)).build(),
            theCallback);
    }

    /**
     * Close the session.
     */
    public void close() {
        theSession.close();
    }

}
```

.NET

```
using PushTechnology.ClientInterface.Client.Content;
using PushTechnology.ClientInterface.Client.Factories;
using PushTechnology.ClientInterface.Client.Features;
using PushTechnology.ClientInterface.Client.Session;
using PushTechnology.ClientInterface.Client.Types;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace PushTechnology.Examples
{
    // A simple client that can send messages to the server or any control
    // client client that has registered a messaging control handler on
 the
    // topic path of the message.
    public class ClientSendingMessages
    {
        private ISession session;
        private IMessaging messaging;
        private ISendCallback callback;
        private ISendContextCallback<String> contextCallback;
```

```
        public ClientSendingMessages(ISendCallback callback,
 ISendContextCallback<String> contextCallback)
        {
            this.callback = callback;
            this.contextCallback = contextCallback;
        }

        public void Open()
        {
            session = Diffusion.Sessions
                .ConnectionTimeout(1200)
                .InputBufferSize(100000)
                .Open("dpt://localhost:8081");
            session.Start();

            messaging = session.GetMessagingFeature();
        }

        public void Close()
        {
            session.Close();
        }

        // Send a message from the client.
        // topicPath - The topic to send the message on
        // message - The payload of the message
        public void Send(string topicPath, string message)
        {
            messaging.Send(topicPath,
 Diffusion.Content.NewContent(message), callback);
        }

        // Send a message from the client using a context for the
 callback.
        // topicPath - The topic to send the message on
        // message - The payload of the message
        // context - The context to be returned in the callback
        public void Send(string topicPath, string message, string context)
        {
            messaging.Send(topicPath,
 Diffusion.Content.NewContent(message), context, contextCallback);
        }

        // Send a message from the client with custom headers.
        // topicPath - The topic to send the message on
        // message - The payload of the message
        // headers - The headers of the message
        public void SendWithHeaders(string topicPath, string message,
 IHeaders headers)
        {
            var sendOptions =
 messaging.CreateSendOptionsBuilder().SetHeaders(headers).Build();
            messaging.Send(topicPath,
 Diffusion.Content.NewContent(message), sendOptions, callback);
        }
    }
}
```

C

```
#include <stdio.h>
#include <unistd.h>

#include "diffusion.h"
#include "args.h"
```

```
ARG_OPTS_T arg_opts[] = {
        ARG_OPTS_HELP,
        {'u', "url", "Diffusion server URL", ARG_OPTIONAL, ARG_HAS_VALUE,
 "dpt://localhost:8081"},
        {'t', "topic", "Topic name", ARG_REQUIRED, ARG_HAS_VALUE, "echo"},
        {'d', "data", "Data to send", ARG_REQUIRED, ARG_HAS_VALUE, NULL},
        END_OF_ARG_OPTS
};

/**
 * Callback invoked when/if a message is published on the topic that the
 * client is writing to.
 */
static int
topic_handler(SESSION_T *session, const TOPIC_MESSAGE_T *msg)
{
        printf("Received message for topic %s\n", msg->name);
        printf("Payload: %.*s\n", (int)msg->payload->len, msg->payload-
>data);
 return HANDLER_SUCCESS;
}

int
main(int argc, char **argv)
{
 // Standard command line parsing.
        HASH_T *options = parse_cmdline(argc, argv, arg_opts);
        if(options == NULL || hash_get(options, "help") != NULL) {
                show_usage(argc, argv, arg_opts);
                return 1;
        }

        char *url = hash_get(options, "url");
        char *topic = hash_get(options, "topic");

 // For subscription, we create a selector which specifies the topic
 // we're sending to *only*.
        char *topic_selector = malloc(1 + strlen(topic) + 1);
        topic_selector[0] = '>';
        strcpy(&topic_selector[1], topic);

 // Create a session with Diffusion.
        SESSION_T *session = NULL;
        DIFFUSION_ERROR_T error;
        session = session_create(url, NULL, NULL, NULL, NULL, &error);
        if(session == NULL) {
                fprintf(stderr, "TEST: Failed to create session\n");
                fprintf(stderr, "ERR : %s\n", error.message);
                return 1;
        }

 // Install a subscription handler before we start the session.
 SUBSCRIPTION_HANDLERS_T *handlers = calloc(1,
 sizeof(SUBSCRIPTION_HANDLERS_T));
 handlers->on_topic_message = &topic_handler;
 subscribe(session, topic_selector, handlers);

 // OK, start the session now.
        if(! session_start(session, &error)) {
                fprintf(stderr, "ERR: Failed to start session: %s\n",
 error.message);
                return 1;
        }

 // Create a topic message.
        TOPIC_MESSAGE_T *msg = calloc(1, sizeof(TOPIC_MESSAGE_T));
        msg->type = MESSAGE_TYPE_TOPIC_LOAD;
```

```
        msg->headers = list_create();
        list_append_last(msg->headers, hash_get(options, "topic"));
        msg->payload = buf_create();
        char *data = hash_get(options, "data");
        buf_write_bytes(msg->payload, data, strlen(data));

 // Send the message.
        send_msg(session, msg, MESSAGE_PRIORITY_MEDIUM, NULL);

 // Wait a few seconds; send_msg is asynchronous and it we don't want
 // to kill the session until the message has been given chance to be
 // sent. We also want to wait for any possible responses.
        sleep(5);

 // Politely close the client connection.
        session_close(session, &error);

        return 0;
}
```

MessagingControl

Use the MessagingControl feature to enable a control client to send messages to specific client sessions and receive messages sent by clients to topics.

The MessagingControl feature is available in the following APIs:

Feature	Java	.NET	C
MessagingControl	YES	YES	PARTIAL Does not support sending messages to a specific client.

The MessagingControl feature contains the following interfaces:

`MessageReceiver`

Implement this interface to provide a handler that receives messages from other clients. The Unified API provides a default implementation that logs the callback.

The following examples use the Unified API to receive messages sent to topics and to send messages to one or more clients :

Java

```
package com.pushtechnology.diffusion.examples;

import com.pushtechnology.diffusion.client.Diffusion;
import com.pushtechnology.diffusion.client.content.Content;
import
 com.pushtechnology.diffusion.client.features.control.topics.MessagingControl;
import
 com.pushtechnology.diffusion.client.features.control.topics.MessagingControl.Message
import
 com.pushtechnology.diffusion.client.features.control.topics.MessagingControl.SendCal
import com.pushtechnology.diffusion.client.session.Session;
import com.pushtechnology.diffusion.client.session.SessionId;
import com.pushtechnology.diffusion.client.types.ReceiveContext;

/**
 * This is an example of a control client using the 'MessagingControl'
 feature
 * to receive messages from clients and also send messages to clients.
```

```
 *
 * It is a trivial example that simply responds to all messages on a
 particular
 * branch of the topic tree by echoing them back to the client exactly as
 they
 * are complete with headers.
 *
 * @author Push Technology Limited
 */
public class ControlClientReceivingMessages {

    private final Session theSession;
    private final MessagingControl theMessagingControl;
    private final SendCallback theSendCallback;

    /**
     * Constructor.
     */
    public ControlClientReceivingMessages() {

        // Set up a default callback which just logs
        theSendCallback = new MessagingControl.SendCallback.Default();

        // Create the session and get the MessagingControl feature
        theSession = Diffusion.sessions().open("dpt://localhost:8081");
        theMessagingControl = theSession.feature(MessagingControl.class);

        // And register to receive all messages sent by clients on the
 "foo"
        // branch
        theMessagingControl.addMessageHandler("foo", new EchoHandler());

        theSession.start();
    }

    /**
     * Close the session.
     */
    public void close() {
        theSession.close();
    }

    /**
     * Handler that echoes messages back to the originating client
 complete with
     * original headers.
     */
    private class EchoHandler extends MessageHandler.Default {
        @Override
        public void onMessage(
            SessionId sessionId,
            String topicPath,
            Content content,
            ReceiveContext context) {

            theMessagingControl.send(
                sessionId,
                topicPath,
                content,
                theMessagingControl.sendOptionsBuilder()
                    .headers(context.getHeaders())
                    .build(),
                theSendCallback);

        }
    }
```

```
}
```

.NET

```
using PushTechnology.ClientInterface.Client.Content;
using PushTechnology.ClientInterface.Client.Factories;
using PushTechnology.ClientInterface.Client.Features;
using PushTechnology.ClientInterface.Client.Features.Control.Topics;
using PushTechnology.ClientInterface.Client.Session;
using PushTechnology.ClientInterface.Client.Types;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace PushTechnology.Examples
{
    // A control client that receives messages on the topic "foo" and
    // echos them back to the client that sent it.
    public class ClientReceivingMessages
    {
        private ISession session;

        public ClientReceivingMessages()
        {
        }

        public void Open()
        {
            // Open the control session
            session = Diffusion.Sessions
                .ConnectionTimeout(1200)
                .InputBufferSize(100000)
                .Open("dpt://localhost:8081");
            session.Start();

            // Set up the feature and handler
            IMessagingControl messagingControl =
 session.GetMessagingControlFeature();
            ISendCallback callback = new NoOpSendCallback();
            messagingControl.AddMessageHandler("foo", new
 EchoHandler(messagingControl, callback));
        }

        public void Close()
        {
            session.Close();
        }
    }

    // A message handler that echos any messages received back to the
 source.
    class EchoHandler : IMessageHandler
    {
        private IMessagingControl messagingControl;
        private ISendCallback callback;

        public EchoHandler(IMessagingControl messagingControl,
 ISendCallback callback)
        {
            this.messagingControl = messagingControl;
            this.callback = callback;
        }

        public void OnMessage(
            SessionId sessionId,
```

```
            String topicPath,
            IContent content,
            IReceiveContext context)
        {
            // Echo the message back to the client that sent it
            messagingControl.Send(
                sessionId,
                topicPath,
                content,
                messagingControl.CreateSendOptionsBuilder()
                    .SetHeaders(context.Headers)
                    .Build(),
                callback);
        }

        public void OnActive(string topicPath, IRegisteredHandler
 registeredHandler)
        {
            // Actions to take before the messages are received by the
 control client
        }

        public void OnClose(string topicPath)
        {
            // Actions to take after the handler is closed
        }
    }

    // No-op send callback that ignores success or failure of the send
    class NoOpSendCallback : ISendCallback
    {
        public void OnComplete()
        {
        }

        public void OnDiscard()
        {
        }
    }
}
```

Related Links

Messaging from a control client on page 238
A control client can use the MessagingControl feature of the Unified API to send individual messages to any known client on any topic. It can also register a handler for messages sent from clients.

Pings

Use the Pings feature to enable a client session to ping the server and verify the connection between client and server.

The Pings feature is available in the following APIs:

Feature	Java	.NET	C
Pings	YES	YES	YES

The Pings feature contains the following classes and interfaces:

`PingCallback`

Implement this interface to provide a callback that receives replies from ping methods. The Unified API provides a default implementation that logs the callback.

PingContextCallback

Implement this interface to provide a callback that receives replies from ping methods. An object can be associated with this callback that provides the context of the call. The Unified API provides a default implementation that logs the callback.

PingDetails

This contains the details associated with a response from a ping to the server.

The following examples use the Unified API to ping the server from a client:

Java

```
package com.pushtechnology.diffusion.examples;

import static org.slf4j.LoggerFactory.getLogger;

import org.slf4j.Logger;

import com.pushtechnology.diffusion.client.Diffusion;
import com.pushtechnology.diffusion.client.features.Pings;
import
 com.pushtechnology.diffusion.client.features.Pings.PingContextCallback;
import com.pushtechnology.diffusion.client.features.Pings.PingDetails;
import com.pushtechnology.diffusion.client.session.Session;

/**
 * This is a simple client example that pings the server and prints out
 the
 * round-trip time.
 *
 * This uses the 'Pings' feature only.
 *
 * @author Push Technology Limited
 */
public final class ClientUsingPings {

    private final Session theSession;
    private final Pings thePings;

    /**
     * Constructor.
     */
    public ClientUsingPings() {
        // Create the session and get the Pings feature
        theSession = Diffusion.sessions().open("dpt://localhost:8081");
        thePings = theSession.feature(Pings.class);
        theSession.start();
    }

    /**
     * Ping the server and log the round trip time.
     *
     * @param context string to log with round trip time
     */
    public void ping(String context) {
        thePings.pingServer(context, new PingLogger());
    }

    /**
     * Close the session.
     */
    public void close() {
```

```
        theSession.close();
    }

    private static final class PingLogger implements
        PingContextCallback<String> {

        private static final Logger LOG = getLogger(PingLogger.class);

        @Override
        public void onPingResponse(String context, PingDetails details) {
            LOG.info(
                "{} - Ping Round Trip Time = {}ms",
                context,
                details.getRoundTripTime());

        }

        @Override
        public void onDiscard(String context) {
            LOG.warn("{} - Ping discarded", context);
        }

    }
}
```

.NET

```
using System;
using PushTechnology.ClientInterface.Client.Session;
using PushTechnology.ClientInterface.Client.Features;
using PushTechnology.ClientInterface.Client.Factories;

namespace PushTechnology.Examples
{

    // A simple client that starts and uses the Pings feature to ping the
    // server.
    class ClientUsingPings
    {
        private ISession session;
        private IPingCallback pingsCallback;
        private IPings pingsFeature;

        public ClientUsingPings()
        {
            // Get a callback that the ping response is returned on
            // The default callback logs the pings response
            pingsCallback = new PingCallbackDefault();
        }

        public void Open()
        {
            session = Diffusion.Sessions
                .ConnectionTimeout(1200)
                .InputBufferSize(100000)
                .Open("dpt://localhost:8081");
            session.Start();

            // Use the session to get the feature
            pingsFeature = session.GetPingFeature();
        }

        // Ping the server the session is connected to.
        public void PingServer()
        {
            // Use the feature to ping the server
```

```
            pingsFeature.PingServer(pingsCallback);
        }

        public void Close()
        {
            session.Close();
        }
    }
}
```

C

```
#include <stdio.h>
#include <unistd.h>

#include "diffusion.h"
#include "args.h"

ARG_OPTS_T arg_opts[] = {
        ARG_OPTS_HELP,
        {'u', "url", "Diffusion server URL", ARG_OPTIONAL, ARG_HAS_VALUE,
 "dpt://localhost:8081"},
        END_OF_ARG_OPTS
};

/**
 * Callback for displaying the receipts of a ping response.
 */
static int
on_ping_response(SESSION_T *session, const SVC_PING_RESPONSE_T *response)
{
 printf("Received ping response\n");
 return HANDLER_SUCCESS;
}

int
main(int argc, char **argv)
{
 // Standard command line parsing.
        HASH_T *options = parse_cmdline(argc, argv, arg_opts);
        if(options == NULL || hash_get(options, "help") != NULL) {
                show_usage(argc, argv, arg_opts);
                return 1;
        }

        char *url = hash_get(options, "url");

 // Create a session with the Diffusion server.
        SESSION_T *session;
        DIFFUSION_ERROR_T error;
        session = session_create(url, NULL, NULL, NULL, NULL, &error);
        if(session == NULL) {
                fprintf(stderr, "Failed to create session: %s\n",
 error.message);
                return 1;
        }

 // Start the session, so we can receive responses to our fetch requests.
        if(! session_start(session, &error)) {
                fprintf(stderr, "Failed to start session: %s\n",
 error.message);
                return 1;
        }

 // Define a ping response handler
 PING_HANDLERS_T *handlers = calloc(1, sizeof(PING_HANDLERS_T));
```

```
 handlers->on_ping_response = &on_ping_response;

 // Send 5 pings at 1 sec intervals
 int i;
 for(i = 0; i < 5; i++) {
  ping(session, handlers);
  sleep(1);
 }

 // Gracefully close the client session.
        session_close(session, &error);

        return 0;
}
```

Security

Use the Security feature to enable a client to change the principal associated with a session.

The Security feature is available in the following APIs:

Feature	Java	.NET	C
Security	YES	YES	YES

The Security feature contains the following classes and interfaces:

`ChangePrincipalCallback`

Implement this interface to provide a callback that receives replies from calls to change a session's principal. The Unified API provides a default implementation that logs the callback.

`ChangePrincipalContextCallback`

Implement this interface to provide a callback that receives replies from calls to change a session's principal. An object can be associated with this callback that provides the context of the call. The Unified API provides a default implementation that logs the callback.

The following examples use the Unified API to change the client principal and credentials for a client session:

Java

```
package com.pushtechnology.diffusion.examples.auth;

import com.pushtechnology.diffusion.client.Diffusion;
import com.pushtechnology.diffusion.client.features.Security;
import
 com.pushtechnology.diffusion.client.features.Security.ChangePrincipalCallback;
import com.pushtechnology.diffusion.client.session.Session;
import com.pushtechnology.diffusion.client.types.Credentials;

/**
 * This demonstrates a client's use of credentials, specifically the
 ability to
 * change credentials for an active session.
 *
 * This is not a realistic use case on its own, but is shown separately
 here for
 * clarity.
 *
 * @author Push Technology Limited
 */
public class ClientUsingCredentials {
```

```
    private final Session theSession;
    private final Security theSecurity;
    private final ChangePrincipalCallback theCallback;

    /**
     * Constructor.
     */
    public ClientUsingCredentials() {

        // Set up a callback that simply logs
        theCallback = new ChangePrincipalCallback.Default();

        // Create the session
        theSession = Diffusion.sessions().open("dpt://localhost:8081");
        theSecurity = theSession.feature(Security.class);
        theSession.start();
    }

    /**
     * Send a change of credentials to the server.
     *
     * @param name the session name
     * @param credentials the credentials
     */
    public void changeCredentials(String name, Credentials credentials) {
        theSecurity.changePrincipal(name, credentials, theCallback);
    }

    /**
     * Close.
     */
    public void close() {
        theSession.close();
    }

}
```

.NET

```
using System;
using PushTechnology.ClientInterface.Client.Session;
using PushTechnology.ClientInterface.Client.Features;
using PushTechnology.ClientInterface.Client.Types;
using PushTechnology.ClientInterface.Client.Factories;

namespace PushTechnology.Examples
{

    //  A simple client that starts and uses the Security feature
    //  to change the credentials associated with the client session.
    class ClientUsingSecurity
    {
        private ISession session;
        private IChangePrincipalCallback changePrincipalCallback;
        private ISecurity securityFeature;

        public ClientUsingSecurity()
        {

            // Get a callback that the response to a change of principal
 is returned on
            // The default callback logs the change of principal response
            changePrincipalCallback = new ChangePrincipalDefault();
        }
```

```
        public void Open()
        {
            // The client starts a session with no associated principal
            session = Diffusion.Sessions
                .ConnectionTimeout(1200)
                .InputBufferSize(100000)
                .Open("dpt://localhost:8081");

            session.Start();

            // Use the session to get the Security feature
            securityFeature = session.GetSecurityFeature();
        }

        // Set the principal and credentials of the connected session.
        // principal - The principal of the session
        // password - A string based password to use as the credentials
        public void SetPrincipalAndPassword(string principal, string
 password)
        {
            // Credentials can be either empty, a String password, or a
 custom sequence of bytes
            // You can use a credentials factory to create a credentials
 object
            ICredentials credentials =
 Diffusion.Credentials.CreatePasswordCredentials(password);

            // Use the Security feature to change the principal
            securityFeature.ChangePrincipal(principal, credentials,
 changePrincipalCallback);
        }

        public void Close()
        {
            session.Close();
        }
    }
}
```

C

```
#include &stdio.h>
#include &stdlib.h>
#include &unistd.h>

#include "diffusion.h"
#include "args.h"

ARG_OPTS_T arg_opts[] = {
        ARG_OPTS_HELP,
        {'u', "url", "Diffusion server URL", ARG_OPTIONAL, ARG_HAS_VALUE,
 "dpt://localhost:8081"},
        {'p', "principal", "Principal (username) for the connection",
 ARG_OPTIONAL, ARG_HAS_VALUE, NULL},
 {'c', "credentials", "Credentials (password) for the connection",
 ARG_OPTIONAL, ARG_HAS_VALUE, NULL},
        END_OF_ARG_OPTS
};

/*
 * Callback to display that the change_principal() request has been
 processed
 * by Diffusion.
 */
static int
```

```
on_change_principal(SESSION_T *session, const
 SVC_CHANGE_PRINCIPAL_RESPONSE_T *response)
{
 printf("on_change_principal\n");
 return HANDLER_SUCCESS;
}

int
main(int argc, char** argv)
{
 // Standard command line parsing.
        HASH_T *options = parse_cmdline(argc, argv, arg_opts);
        if(options == NULL || hash_get(options, "help") != NULL) {
                show_usage(argc, argv, arg_opts);
                return 1;
        }

 char *url = hash_get(options, "url");

 // Create a session with Diffusion, with no principal or credentials.
 SESSION_T *session;
 DIFFUSION_ERROR_T error;
 session = session_create(url, NULL, NULL, NULL, NULL, &error);
 if(session == NULL) {
  fprintf(stderr, "Failed to create session: %s\n", error.message);
  return 1;
 }

 // Start the session.
 if(! session_start(session, &error)) {
  fprintf(stderr, "Failed to start session: %s\n", error.message);
  return 1;
 }

 puts("Session started");

 // Wait for a couple of seconds.
 sleep(2);

 puts("Changing credentials");

 // Specify callbacks for the change_principal request.
 CHANGE_PRINCIPAL_HANDLERS_T *handlers = calloc(1,
 sizeof(CHANGE_PRINCIPAL_HANDLERS_T));
 handlers->on_change_principal = on_change_principal;

 // Do the change.
 CREDENTIALS_T *credentials = credentials_create_password("chips");
 change_principal(session, "fish", credentials, handlers);

 // Wait for a couple more seconds.
 sleep(2);

 puts("Closing session");

 // Gracefully close the connection.
 session_close(session, &error);

 return 0;
}
```

SubscriptionControl

Use the SubscriptionControl feature to enable a control client to subscribe other clients to topics and handle routing topic subscription requests.

The SubscriptionControl feature is available in the following APIs:

Feature	Java	.NET	C
SubscriptionControl	YES	YES	NO

The SubscriptionControl feature contains the following classes and interfaces:

`RoutingSubscriptionRequest`
: Implement this interface to provide a request to subscribe to a routing topic.

`RoutingSubscriptionRequest.Handler`
: Implement this interface to provide a handler for responses to requests to subscribe to a routing topic.

`SubscriptionCallback`
: Implement this interface to provide a callback that receives status notifications from subscription calls. The Unified API provides a default implementation that logs the callback.

`SubscriptionContextCallback`
: Implement this interface to provide a callback that receives status notifications from subscription calls. An object can be associated with this callback that provides the context of the call. The Unified API provides a default implementation that logs the callback.

The example uses the Unified API to notify a control client when another client requests a subscription to a routing topic:

Java

```
package com.pushtechnology.diffusion.examples;

import com.pushtechnology.diffusion.client.Diffusion;
import
 com.pushtechnology.diffusion.client.features.control.topics.SubscriptionControl;
import
 com.pushtechnology.diffusion.client.features.control.topics.SubscriptionControl.Rout
import
 com.pushtechnology.diffusion.client.features.control.topics.SubscriptionControl.Subs
import com.pushtechnology.diffusion.client.session.Session;
import com.pushtechnology.diffusion.client.session.SessionId;

/**
 * This demonstrates using a control client to be notified of subscription
 * requests to routing topics and to subscribe/unsubscribe clients to
 topics.
 *
 * This uses the 'SubscriptionControl' feature.
 *
 * @author Push Technology Limited
 */
public class ControlClientSubscriptionControl {

    private final Session theSession;

    private SubscriptionControl theSubscriptionControl;
```

```
    private SubscriptionCallback theSubscriptionCallback;

    /**
     * Constructor.
     */
    public ControlClientSubscriptionControl() {

        // Set up a default callback that just logs
        theSubscriptionCallback = new SubscriptionCallback.Default();

        // Create the session
        theSession = Diffusion.sessions().open("dpt://localhost:8081");

        theSubscriptionControl =
theSession.feature(SubscriptionControl.class);

        // Sets up a handler so that all subscriptions to topic a/b are
routed
        // to routing/target/topic
        theSubscriptionControl.addRoutingSubscriptionHandler(
            "a/b",
            new SubscriptionControl.RoutingSubscriptionRequest.Handler
            .Default() {
                @Override
                public void onSubscriptionRequest(
                    final RoutingSubscriptionRequest request) {

                    request.route(
                        "routing/target/topic",
                        theSubscriptionCallback);
                }

            });

        theSession.start();
    }

    /**
     * Subscribe a client to topics.
     *
     * @param sessionId client to subscribe
     * @param topicSelector topic selector expression
     */
    public void subscribe(SessionId sessionId, String topicSelector) {
        theSubscriptionControl.subscribe(
            sessionId,
            topicSelector,
            theSubscriptionCallback);
    }

    /**
     * Unsubscribe a client from topics.
     *
     * @param sessionId client to unsubscribe
     * @param topicSelector topic selector expression
     */
    public void unsubscribe(SessionId sessionId, String topicSelector) {
        theSubscriptionControl.unsubscribe(
            sessionId,
            topicSelector,
            theSubscriptionCallback);
    }

    /**
     * Close the session.
     */
```

```
    public void close() {
        theSession.close();
    }
}
```

.NET

```
using PushTechnology.ClientInterface.Client.Factories;
using PushTechnology.ClientInterface.Client.Features.Control.Topics;
using PushTechnology.ClientInterface.Client.Session;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace PushTechnology.Examples
{
    // A control client handling subscription requests for routing topic
 data.
    public class ControlClientSubscriptionControl
    {
        private ISubscriptionCallback callback;
        private ISession session;
        private ISubscriptionControl subscriptionControl;

        public ControlClientSubscriptionControl()
        {
            callback = new SubscriptionCallbackDefault();
        }

        public void Open()
        {
            session = Diffusion.Sessions
                .ConnectionTimeout(1200)
                .InputBufferSize(100000)
                .Open("dpt://localhost:8081");

            subscriptionControl = session.GetSubscriptionControlFeature();

            subscriptionControl.AddRoutingSubscriptionHandler("a/b", new
 SubscriptionRequestHandler(callback));

            session.Start();
        }

        public void Close()
        {
            session.Close();
        }


        // Subscribe a session to a topic selector
        // sessionId - The session to subscribe
        // topicSelector - The selector to subscribe to
        public void Subscribe(SessionId sessionId, String topicSelector)
        {
            subscriptionControl.Subscribe(sessionId, topicSelector,
 callback);
        }

        // Unsubscribe a session from a topic selector
        // sessionId - The session to unsubscribe
        // topicSelector - The selector to unsubscribe from
        public void Unsubscribe(SessionId sessionId, String topicSelector)
        {
```

```
            subscriptionControl.Unsubscribe(sessionId, topicSelector,
 callback);
        }
    }


    // A subscription request handler for routing topics.
    class SubscriptionRequestHandler : IRoutingSubscriptionRequestHandler
    {
        private ISubscriptionCallback callback;

        public SubscriptionRequestHandler(ISubscriptionCallback callback)
        {
            this.callback = callback;
        }

        public void OnSubscriptionRequest(IRoutingSubscriptionRequest
 request)
        {
            // Route the subscription request to the topic path "routing/
target/topic"
            request.Route("routing/target/topic", callback);
        }

        public void OnActive(string topicPath,
 ClientInterface.Client.Features.IRegisteredHandler registeredHandler)
        {
            // Actions to take before the subscription requests are
 received by the control client
        }

        public void OnClose(string topicPath)
        {
            // Actions to take after the handler is closed
        }
    }
}
```

Related Links

Managing subscriptions from a control client on page 236
A control client can use the SubscriptionControl feature of the Unified API to subscribe other client sessions to topics that they have not requested subscription to themselves and also to unsubscribe clients from topics. It also enables the control client to register as the handler for routing topic subscriptions.

Control client on page 226
A Diffusion client application can take on the role of control client.

TopicControl

Use the TopicControl feature to enable a control client session to create and manage topics and to provide state for delegated topics.

The TopicControl feature is available in the following APIs:

Feature	Java	.NET	C
TopicControl	YES	YES	PARTIAL Does not support paged topics

The TopicControl feature contains the following classes and interfaces:

`AddCallback`

Implement this interface to provide a callback that receives responses from add topic calls. The Unified API provides a default implementation that logs the callback.

`AddContextCallback`

Implement this interface to provide a callback that receives responses from add topic calls. An object can be associated with this callback that provides the context of the call. The Unified API provides a default implementation that logs the callback.

`MissingTopicHandler`

Implement this interface to provide a handler that is called when a client session makes a subscription or fetch request for a topic that does not exist. The Unified API provides a default implementation that logs the missing topic events.

`MissingTopicNotification`

Implement this interface to provide the notification that occurs when a client session makes a subscription or fetch request for a topic that does not exist.

`RemoveCallback`

Implement this interface to provide a callback that receives responses from remove topic calls. The Unified API provides a default implementation that logs the callback.

`RemoveContextCallback`

Implement this interface to provide a callback that receives responses from remove topic calls. An object can be associated with this callback that provides the context of the call. The Unified API provides a default implementation that logs the callback.

`StateProvider`

Implement this interface to notify of a request for the state of a delegated topic.

`StateProvider.Request`

Implement this interface to request the state of a delegated topic.

Related Links

Maintaining topics from a control client on page 230
A control client can use the TopicControl feature of the Unified API to add and remove topics at the server.

Creating paged topics on page 231
Use the `TopicDetails.Builder` classes with the TopicControl feature to create paged topics.

Removing topics with sessions on page 233
A control client can use the TopicControl feature of the Unified API to specify that the Diffusion server removes a topic or topics after the control client session closes or fails.

Control client on page 226

A Diffusion client application can take on the role of control client.

Example: Creating a topic

The following examples use the TopicControl feature in the Unified API to create topics.

Java

```
package com.pushtechnology.diffusion.examples;

import com.pushtechnology.diffusion.client.Diffusion;
import com.pushtechnology.diffusion.client.callbacks.TopicTreeHandler;
import com.pushtechnology.diffusion.client.content.Content;
import com.pushtechnology.diffusion.client.content.metadata.MContent;
import
 com.pushtechnology.diffusion.client.features.control.topics.TopicControl;
import
 com.pushtechnology.diffusion.client.features.control.topics.TopicControl.AddContextC
import
 com.pushtechnology.diffusion.client.features.control.topics.TopicControl.RemoveCallb
import com.pushtechnology.diffusion.client.session.Session;
import
 com.pushtechnology.diffusion.client.topics.details.RecordTopicDetails;
import com.pushtechnology.diffusion.client.topics.details.TopicDetails;
import com.pushtechnology.diffusion.client.topics.details.TopicType;

/**
 * An example of using a control client to add topics.
 *
 * This uses the 'TopicControl' feature only.
 *
 * @author Push Technology Limited
 */
public class ControlClientAddingTopics {

    private final Session theSession;

    private final TopicControl theTopicControl;

    private final AddContextCallback<String> theAddCallback;

    private final RemoveCallback theRemoveCallback;

    private final TopicDetails theStringTopicDetails;

    /**
     * Constructor.
     *
     * @param callback an object o call back on to report the success or
 failure
     *        of each topic add request
     */
    public ControlClientAddingTopics(AddContextCallback<String> callback)
 {

        theAddCallback = callback;

        // Just use a default callback for removals
        theRemoveCallback = new TopicControl.RemoveCallback.Default();

        // Create the session
        theSession = Diffusion.sessions().open("dpt://localhost:8080");

        theTopicControl = theSession.feature(TopicControl.class);

        // For details that may be reused may times it is far more
 efficient to
```

```
        // create just once - this creates a default string type details.
        theStringTopicDetails =
            theTopicControl.newDetails(TopicType.SINGLE_VALUE);

        theSession.start();
    }

    /**
     * Adds a single value string topic.
     *
     * @param topicPath full topic path
     * @param context this will be passed back to the callback when
reporting
     *        success or failure of the topic add
     * @param initialValue an optional initial value for the topic
     */
    public void addStringTopic(
        String topicPath, String context, String initialValue) {

        Content content = null;
        if (initialValue != null) {
            content = Diffusion.content().newContent(initialValue);
        }

        theTopicControl.addTopic(
            topicPath,
            theStringTopicDetails,
            content,
            context,
            theAddCallback);
    }

    /**
     * Adds a record topic with supplied metadata and optional initial
content.
     *
     * This example shows details being created and would be fine when
creating
     * topics that are all different but if creating many record topics
with the
     * same details then it is far more efficient to pre-create the
details.
     *
     * @param topicPath the full topic path
     * @param context context passed back to callback when topic created
     * @param metadata pre-created record metadata
     * @param initialValue optional initial value for the topic which must
have
     *        been created to match the supplied metadata
     */
    public void addRecordTopic(
        String topicPath,
        String context,
        MContent metadata,
        Content initialValue) {

        final TopicDetails details =

theTopicControl.newDetailsBuilder(RecordTopicDetails.Builder.class)
                .metadata(metadata).build();

        theTopicControl.addTopic(
            topicPath,
            details,
            initialValue,
            context,
            theAddCallback);
```

```
    }

    /**
     * Remove a single topic given its path.
     *
     * @param topicPath the topic path
     */
    public void removeTopic(String topicPath) {
        theTopicControl.removeTopics(
            ">" + topicPath, // convert to a topic path selector
            theRemoveCallback);
    }

    /**
     * Remove one or more topics using a topic selector expression.
     *
     * @param topicSelector the selector expression
     */
    public void removeTopics(String topicSelector) {
        theTopicControl.removeTopics(
            topicSelector,
            theRemoveCallback);
    }

    /**
     * Request that the topic {@code topicPath} and its descendants be
 removed
     * when the session is closed (either explicitly using {@link
 Session#close}
     * , or by the server). If more than one session calls this method for
 the
     * same {@code topicPath}, the topics will be removed when the last
 session
     * is closed.
     *
     * Different sessions may call this method for the same topic path,
 but not
     * for topic paths above or below existing registrations on the same
 branch
     * of the topic tree.
     *
     * @param topicPath the part of the topic tree to remove when the last
     *        session is closed
     */
    public void removeTopicsWithSession(String topicPath) {
        theTopicControl.removeTopicsWithSession(
            topicPath, new TopicTreeHandler.Default());
    }

    /**
     * Close the session.
     */
    public void close() {
        theSession.close();
    }

}
```

.NET

```
using PushTechnology.ClientInterface.Client.Content;
using PushTechnology.ClientInterface.Client.Content.Metadata;
using PushTechnology.ClientInterface.Client.Factories;
using PushTechnology.ClientInterface.Client.Features.Control.Topics;
using PushTechnology.ClientInterface.Client.Session;
using PushTechnology.ClientInterface.Client.Topics;
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace PushTechnology.Examples
{

    /// A control client that uses the ITopicControl feature to create and
    /// remove topics.

    class ControlClientAddingTopics
    {
        private ISession session;
        private ITopicControl topicControl;
        private ITopicControlAddContextCallback<string> addCallback;
        private ITopicControlRemoveCallback removeCallback;
        private ITopicDetails stringTopicDetails;

        public ControlClientAddingTopics()
        {
            addCallback = new
 TopicControlAddContextCallbackDefault<string>();
            removeCallback = new TopicControlRemoveCallbackDefault();
        }

        public void Open()
        {
            session = Diffusion.Sessions
                .ConnectionTimeout(1200)
                .InputBufferSize(100000)
                .Open("dpt://localhost:8081");

            // Use the session to get the feature
            topicControl = session.GetTopicControlFeature();

            // Use the feature to create a single value topic details
            stringTopicDetails =
 topicControl.NewDetails(TopicType.SINGLE_VALUE);

            session.Start();
        }

        public void Close()
        {
            session.Close();
        }

        ///
        /// Add a topic using single value topic data.
        ///
        /// topicPath The name of the topic
        /// context Context to be passed to the callback
        /// initialValue The initial value of the topic
        public void AddStringTopic(string topicPath, string context,
 string initialValue)
        {
            IContent content = null;
            if (initialValue != null)
            {
                // Turn the initial value into an IContent object
                content = Diffusion.Content.NewContent(initialValue);
            }

            // Use the feature to add a topic using the single value topic
            // details and the initial content of the topic
```

```
            topicControl.AddTopic(topicPath, stringTopicDetails, content,
 context, addCallback);
        }

        ///
        /// Add a topic using record topic data.
        ///
        /// topicPath The name of the topic
        /// context Context to be passed to the callback
        /// metadata The metadata to use for the topic
        /// initialValue The initial value of the topic
        public void AddRecordTopic(string topicPath, string context,
 IMContent metadata, IContent initialValue)
        {
            // Create a record topic details out of the metadata of the
 topic
            IRecordTopicDetails details =
 topicControl.CreateDetailsBuilder<IRecordTopicDetailsBuilder>()
                .Metadata(metadata)
                .Build();

            // Add a topic using the record topic details and an initial
 value
            topicControl.AddTopic(topicPath, details, initialValue,
 context, addCallback);
        }

        ///
        /// Remove a topic.
        ///
        /// topicPath The topic to remove
        public void RemoveTopic(string topicPath)
        {
            topicControl.RemoveTopics(">" + topicPath, removeCallback);
        }

        ///
        /// Remove many topics.
        ///
        /// topicSelector  The topic selector that matches the
        /// topics to remove
        public void RemoveTopics(string topicSelector)
        {
            topicControl.RemoveTopics(topicSelector, removeCallback);
        }

        ///
        /// Register a part of the topic tree to remove on disconnection.
        ///
        /// topicPath  The part of the topic tree to remove
        public void RegisterTopicRemoval(string topicPath)
        {
            topicControl.RemoveTopicsWithSession(topicPath, handler);
        }

    }
}
```

C

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <apr.h>
#include <apr_thread_mutex.h>
```

```
#include <apr_thread_cond.h>

#include "diffusion.h"
#include "args.h"
#include "service/svc-topic-control.h"
#include "utils.h"

apr_pool_t *pool = NULL;
apr_thread_mutex_t *mutex = NULL;
apr_thread_cond_t *cond = NULL;

ARG_OPTS_T arg_opts[] = {
        ARG_OPTS_HELP,
        {'u', "url", "Diffusion server URL", ARG_OPTIONAL, ARG_HAS_VALUE,
 "dpt://localhost:8081"},
        END_OF_ARG_OPTS
};

// Various handlers which are common to all add_topic() functions.
static int
on_topic_added(SVC_ADD_TOPIC_RESPONSE_T *response)
{
 puts("on_topic_added");
 apr_thread_cond_broadcast(cond);
 return HANDLER_SUCCESS;
}

static int
on_topic_add_failed(SVC_ADD_TOPIC_RESPONSE_T *response)
{
 printf("on_topic_add_failed: %d\n", response->response_code);
 apr_thread_cond_broadcast(cond);
 return HANDLER_SUCCESS;
}

static int
on_topic_add_discard(SVC_ADD_TOPIC_RESPONSE_T *response)
{
 puts("on_topic_add_discard");
 apr_thread_cond_broadcast(cond);
 return HANDLER_SUCCESS;
}

static int
on_topic_removed(SVC_REMOVE_TOPICS_RESPONSE_T *response)
{
 puts("on_topic_removed");
 apr_thread_cond_broadcast(cond);
 return HANDLER_SUCCESS;
}

static int
on_topic_remove_discard(SVC_REMOVE_TOPICS_RESPONSE_T *response)
{
 puts("on_topic_remove_discard");
 apr_thread_cond_broadcast(cond);
 return HANDLER_SUCCESS;
}
/*
 *
 */
int main(int argc, char** argv)
{
 // Standard command line parsing.
        HASH_T *options = parse_cmdline(argc, argv, arg_opts);
        if(options == NULL || hash_get(options, "help") != NULL) {
                show_usage(argc, argv, arg_opts);
```

```
                return 1;
        }

        char *url = hash_get(options, "url");

// Setup for condition variable
apr_initialize();
apr_pool_create(&pool, NULL);
apr_thread_mutex_create(&mutex, APR_THREAD_MUTEX_DEFAULT, pool);
apr_thread_cond_create(&cond, pool);

// Setup for session
SESSION_T *session;
DIFFUSION_ERROR_T error;
session = session_create(url, NULL, NULL, NULL, NULL, &error);
if(session == NULL) {
                fprintf(stderr, "TEST: Failed to create session\n");
                fprintf(stderr, "ERR : %s\n", error.message);
                return 1;
        }
        if(! session_start(session, &error)) {
                fprintf(stderr, "ERR: Failed to start session: %s\n",
error.message);
                return 1;
        }

// Handlers used by all add_topic() functions.
ADD_TOPIC_HANDLERS_T *add_handlers = calloc(1,
sizeof(ADD_TOPIC_HANDLERS_T));
add_handlers->on_topic_added = on_topic_added;
add_handlers->on_topic_add_failed = on_topic_add_failed;
add_handlers->on_discard = on_topic_add_discard;

// Stateless topic details
TOPIC_DETAILS_T *topic_details = create_topic_details_stateless();
apr_thread_mutex_lock(mutex);
add_topic(session, "stateless", topic_details, NULL, add_handlers);
apr_thread_cond_wait(cond, mutex);
apr_thread_mutex_unlock(mutex);

// Single value string data, no default data.
TOPIC_DETAILS_T *string_topic_details =
create_topic_details_single_value(M_DATA_TYPE_STRING);
apr_thread_mutex_lock(mutex);
add_topic(session, "string", string_topic_details, NULL, add_handlers);
apr_thread_cond_wait(cond, mutex);
apr_thread_mutex_unlock(mutex);

// Single value string data, with default data.
BUF_T *sample_data_buf = buf_create();
buf_write_string(sample_data_buf, "Hello, world");
CONTENT_T *content = content_create(CONTENT_ENCODING_NONE,
sample_data_buf);

apr_thread_mutex_lock(mutex);
add_topic(session, "string-data", string_topic_details, content,
add_handlers);
apr_thread_cond_wait(cond, mutex);
apr_thread_mutex_unlock(mutex);

// Single value integer data.
TOPIC_DETAILS_T *integer_topic_details =
create_topic_details_single_value(M_DATA_TYPE_INTEGER_STRING);
integer_topic_details->topic_details_params.integer.default_value = 999;

apr_thread_mutex_lock(mutex);
add_topic(session, "integer", integer_topic_details, NULL, add_handlers);
```

```
 apr_thread_cond_wait(cond, mutex);
 apr_thread_mutex_unlock(mutex);

 // Single value integer data with content.
 BUF_T *integer_data_buf = buf_create();
 buf_sprintf(integer_data_buf, "%d", 123);
 CONTENT_T *content_integer = content_create(CONTENT_ENCODING_NONE,
 integer_data_buf);

 apr_thread_mutex_lock(mutex);
 add_topic(session, "integer-data", integer_topic_details,
 content_integer, add_handlers);
 apr_thread_cond_wait(cond, mutex);
 apr_thread_mutex_unlock(mutex);

 // Single value decimal data.
 TOPIC_DETAILS_T *decimal_topic_details =
 create_topic_details_single_value(M_DATA_TYPE_DECIMAL_STRING);
 decimal_topic_details->topic_details_params.decimal.default_value =
 123.456;
 decimal_topic_details->topic_details_params.decimal.scale = 4;

 apr_thread_mutex_lock(mutex);
 add_topic(session, "decimal", decimal_topic_details, NULL, add_handlers);
 apr_thread_cond_wait(cond, mutex);
 apr_thread_mutex_unlock(mutex);

 // Single value decimal data with content.
 BUF_T *decimal_data_buf = buf_create();
 buf_sprintf(decimal_data_buf, "%f", 987.654);
 CONTENT_T *content_decimal = content_create(CONTENT_ENCODING_NONE,
 decimal_data_buf);

 apr_thread_mutex_lock(mutex);
 add_topic(session, "decimal-data", decimal_topic_details,
 content_decimal, add_handlers);
 apr_thread_cond_wait(cond, mutex);
 apr_thread_mutex_unlock(mutex);

 // Manually specify schema for single value string topic data.
 BUF_T *manual_schema = buf_create();
 buf_write_string(manual_schema,
  "<?xml version=\"1.0\" encoding=\"UTF-8\" standalone=\"yes\"?>\n");
 buf_write_string(manual_schema,
  "<field name=\"x\" type=\"string\" default=\"xyzzy\" allowsEmpty=\"true
\"/>");
 TOPIC_DETAILS_T *manual_topic_details =
 create_topic_details_single_value(M_DATA_TYPE_STRING);
 manual_topic_details->user_defined_schema = manual_schema;

 apr_thread_mutex_lock(mutex);
 add_topic(session, "string-manual", manual_topic_details, NULL,
 add_handlers);
 apr_thread_cond_wait(cond, mutex);
 apr_thread_mutex_unlock(mutex);

 // Simple record/field structure, created by manually specifying the
 // schema.
 BUF_T *record_schema = buf_create();
 buf_write_string(record_schema,
  "<?xml version=\"1.0\" encoding=\"UTF-8\" standalone=\"yes\"?>");
 buf_write_string(record_schema,
  "<message topicDataType=\"record\" name=\"MyContent\">");
 buf_write_string(record_schema,
  "<record name=\"Record1\">");
 buf_write_string(record_schema,
```

```
  "<field type=\"string\" default=\"\" allowsEmpty=\"true\" name=
\"Field1\"/>");
 buf_write_string(record_schema,
  "<field type=\"integerString\" default=\"0\" allowsEmpty=\"false\" name=
\"Field2\"/>");
 buf_write_string(record_schema,
  "<field type=\"decimalString\" default=\"0.00\" scale=\"2\" allowsEmpty=
\"false\" name=\"Field3\"/>");
 buf_write_string(record_schema,
  "</record>");
 buf_write_string(record_schema,
  "</message>");
 TOPIC_DETAILS_T *record_topic_details = create_topic_details_record();
 record_topic_details->user_defined_schema = record_schema;

 apr_thread_mutex_lock(mutex);
 add_topic(session, "record", record_topic_details, NULL, add_handlers);
 apr_thread_cond_wait(cond, mutex);
 apr_thread_mutex_unlock(mutex);

 /* Remove topic tests */
 puts("Adding topics remove_me/1 and remove_me/2");
 apr_thread_mutex_lock(mutex);
 add_topic(session, "remove_me/1", topic_details, NULL, add_handlers);
 apr_thread_cond_wait(cond, mutex);
 apr_thread_mutex_unlock(mutex);
 add_topic(session, "remove_me/2", topic_details, NULL, add_handlers);
 apr_thread_cond_wait(cond, mutex);
 apr_thread_mutex_unlock(mutex);

 puts("Removing topics in 5 seconds...");
 sleep(5);

 REMOVE_TOPICS_HANDLERS_T *remove_handlers = calloc(1,
 sizeof(REMOVE_TOPICS_HANDLERS_T));
 remove_handlers->on_removed = on_topic_removed;
 remove_handlers->on_discard = on_topic_remove_discard;

 apr_thread_mutex_lock(mutex);
 remove_topics(session, ">remove_me", remove_handlers);
 apr_thread_cond_wait(cond, mutex);
 apr_thread_mutex_unlock(mutex);

 return(EXIT_SUCCESS);
}
```

Related Links

Control client on page 226
A Diffusion client application can take on the role of control client.

Example: Creating a paged topic

The following examples use the TopicControl feature in the Unified API to create paged topics.

Java

```
package com.pushtechnology.diffusion.examples;

import com.pushtechnology.diffusion.client.Diffusion;
import com.pushtechnology.diffusion.client.content.metadata.MRecord;
import
 com.pushtechnology.diffusion.client.content.metadata.MetadataFactory;
import
 com.pushtechnology.diffusion.client.content.update.PagedRecordOrderedUpdateFactory;
import
 com.pushtechnology.diffusion.client.content.update.PagedStringUnorderedUpdateFactory
import com.pushtechnology.diffusion.client.content.update.Update;
```

```
import
 com.pushtechnology.diffusion.client.features.control.topics.TopicControl;
import
 com.pushtechnology.diffusion.client.features.control.topics.TopicControl.AddCallback
import
 com.pushtechnology.diffusion.client.features.control.topics.TopicControl.RemoveCallb
import
 com.pushtechnology.diffusion.client.features.control.topics.TopicUpdateControl;
import
 com.pushtechnology.diffusion.client.features.control.topics.TopicUpdateControl.Updat
import
 com.pushtechnology.diffusion.client.features.control.topics.TopicUpdateControl.Updat
import
 com.pushtechnology.diffusion.client.features.control.topics.TopicUpdateControl.Updat
import com.pushtechnology.diffusion.client.session.Session;
import
 com.pushtechnology.diffusion.client.topics.details.PagedRecordTopicDetails;
import
 com.pushtechnology.diffusion.client.topics.details.PagedRecordTopicDetails.Attribute
import com.pushtechnology.diffusion.client.topics.details.TopicType;

/**
 * An example of using a control client to create and update paged topics.
 *
 * This uses the 'TopicControl' feature to create a paged topic and the
 * 'TopicUpdateControl' feature to send updates to it.
 *
 * This demonstrates some simple examples of paged topic updates but not
 all of
 * the possible ways in which they can be done.
 *
 * @author Push Technology Limited
 */
public class ControlClientUpdatingPagedTopics {

    private static final String ORDERED_TOPIC = "Paged/Ordered";
    private static final String UNORDERED_TOPIC = "Paged/Unordered";

    private final Session theSession;
    private final TopicControl theTopicControl;
    private final TopicUpdateControl theUpdateControl;
    private final UpdateCallback theUpdateCallback;
    private final PagedRecordOrderedUpdateFactory theOrderedUpdateFactory;
    private final PagedStringUnorderedUpdateFactory
 theUnorderedUpdateFactory;
    private Updater theUpdater = null;

    /**
     * Constructor.
     */
    public ControlClientUpdatingPagedTopics() {

        theUpdateCallback = new UpdateCallback.Default();

        // Create the session
        theSession = Diffusion.sessions().open("dpt://localhost:8081");
        theTopicControl = theSession.feature(TopicControl.class);
        theUpdateControl = theSession.feature(TopicUpdateControl.class);

        theOrderedUpdateFactory =
            theUpdateControl
                .updateFactory(PagedRecordOrderedUpdateFactory.class);
        theUnorderedUpdateFactory =
            theUpdateControl
                .updateFactory(PagedStringUnorderedUpdateFactory.class);

        theSession.start();
```

```
        final MetadataFactory metadata = Diffusion.metadata();

        // Create an unordered paged string topic
        theTopicControl.addTopic(
            UNORDERED_TOPIC,
            theTopicControl.newDetails(TopicType.PAGED_STRING),
            new AddCallback.Default());

        // Create an ordered paged record topic
        final MRecord recordMetadata =
            metadata.record(
                "Record",
                metadata.string("Name"),
                metadata.string("Address"));

        theTopicControl.addTopic(
            ORDERED_TOPIC,
            theTopicControl.
                newDetailsBuilder(PagedRecordTopicDetails.Builder.class).
                metadata(recordMetadata).
                order(new OrderKey("Name")).build(),
            new AddCallback.Default());

        // Register as updater for topics under the 'Paged' branch
        theUpdateControl.registerUpdateSource(
            "Paged",
            new UpdateSource.Default() {
                @Override
                public void onActive(String topicPath, Updater updater) {
                    theUpdater = updater;
                }
            });
    }

    /**
     * Add a new line to the ordered topic.
     *
     * @param name the name field value
     * @param address the address field value
     */
    public void addOrdered(String name, String address) {
        update(
            ORDERED_TOPIC,
            theOrderedUpdateFactory.add(
                Diffusion.content().newRecord(name, address)));
    }

    /**
     * Update a line of an ordered topic.
     *
     * @param name the name of the line to update
     * @param address the new address field value
     */
    public void updateOrdered(String name, String address) {
        update(
            ORDERED_TOPIC,
            theOrderedUpdateFactory.update(
                Diffusion.content().newRecord(name, address)));
    }

    /**
     * Remove a line from an ordered topic.
     *
     * @param name the name of the line to remove
     */
    public void removeOrdered(String name) {
```

```
        update(
            ORDERED_TOPIC,
            theOrderedUpdateFactory.remove(
                Diffusion.content().newRecord(name, "")));
    }

    /**
     * Add a line or lines to the end of an unordered topic.
     *
     * @param values lines to add
     */
    public void addUnordered(String... values) {
        update(
            UNORDERED_TOPIC,
            theUnorderedUpdateFactory.add(values));
    }

    /**
     * Insert a line or lines at a specified index within an unordered
topic.
     *
     * @param index the index to add at
     * @param values lines to insert
     */
    public void insertUnordered(int index, String... values) {
        update(
            UNORDERED_TOPIC,
            theUnorderedUpdateFactory.insert(index, values));
    }

    /**
     * Update a line within an unordered topic.
     *
     * @param index the index of the line to update
     * @param value the new line value
     */
    public void updateUnordered(int index, String value) {
        update(
            UNORDERED_TOPIC,
            theUnorderedUpdateFactory.update(index, value));
    }

    /**
     * Remove a specific line from an unordered topic.
     *
     * @param index the line to remove
     */
    public void removeUnordered(int index) {
        update(
            UNORDERED_TOPIC,
            theUnorderedUpdateFactory.remove(index));
    }

    private void update(String topic, Update update)
        throws IllegalStateException {
        if (theUpdater == null) {
            throw new IllegalStateException("No updater");
        }
        theUpdater.update(topic, update, theUpdateCallback);
    }

    /**
     * Close the session.
     */
    public void close() {
        // Remove our topic and close session when done
        theTopicControl.removeTopics(
```

```
            ">Paged",
            new RemoveCallback() {
                @Override
                public void onDiscard() {
                    theSession.close();
                }

                @Override
                public void onTopicsRemoved() {
                    theSession.close();
                }
            });

    }
}
```

.NET

```
using PushTechnology.ClientInterface.Client.Content.Metadata;
using PushTechnology.ClientInterface.Client.Factories;
using PushTechnology.ClientInterface.Client.Features.Control.Topics;
using PushTechnology.ClientInterface.Client.Session;
using PushTechnology.ClientInterface.Client.Topics;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace PushTechnology.Examples
{
    ///
    /// A control client that uses the ITopicControl feature to create
 paged
    /// topics.
    ///
    class ControlClientAddingPagedTopics
    {
        private const string ORDERED_TOPIC = "Paged/Ordered";
        private const string UNORDERED_TOPIC = "Paged/Unordered";

        private ISession session;
        private ITopicControl topicControl;
        private ITopicControlAddCallback addCallback;
        private ITopicControlRemoveCallback removeCallback;

        public ControlClientAddingPagedTopics()
        {
            addCallback = new TopicControlAddCallbackDefault();
            removeCallback = new TopicControlRemoveCallbackDefault();
        }

        public void Open()
        {
            session = Diffusion.Sessions
                .ConnectionTimeout(1200)
                .InputBufferSize(100000)
                .Open("dpt://localhost:8081");

            // Use the session to get the feature
            topicControl = session.GetTopicControlFeature();

            session.Start();
        }

        public void Close()
        {
```

```
            session.Close();
        }

        private void CreatePagedStringTopic()
        {
            // Use the feature to create a paged string topic
            topicControl.AddTopic(UNORDERED_TOPIC,
topicControl.NewDetails(TopicType.PAGED_STRING), addCallback);
        }

        private void AddOrderedPagedRecordTopic()
        {
            IMetadataFactory metadata = Diffusion.Metadata;
            // Create the metadata for the topic messages
            IMRecord recordMetadata = metadata
                .Record("record",
                    metadata.String("Name"),
                    metadata.String("Address"));

            // Create a paged record ordered topic details
            IPagedRecordTopicDetails details =
topicControl.CreateDetailsBuilder<IPagedRecordTopicDetailsBuilder>()
                .Metadata(recordMetadata)
                .Order(new PagedRecordOrderKey("Name")) // Order the lines
by the field names "Name"
                .Build();

            // Use the feature to create a topic from the topic details
            topicControl.AddTopic(ORDERED_TOPIC, details, addCallback);
        }
    }
}
```

Related Links

Control client on page 226
A Diffusion client application can take on the role of control client.

Topics

Use the Topics feature to enable a client session to subscribe to topics and receive streaming update to those topics or to fetch the current state of topics.

The Topics feature is available in the following APIs:

Feature	Java	.NET	C
Topics	YES	YES	YES

The Topics feature contains the following classes and interfaces:

`CompletionCallback`

Implement this interface to provide a callback that receives success or failure notifications from calls. The Unified API provides a default implementation that logs the callback.

`CompletionContextCallback`

Implement this interface to provide a callback that receives success or failure notifications from calls. An object can be associated with this callback that provides the context of the call. The Unified API provides a default implementation that logs the callback.

FetchStream

Implement this interface to provide a callback that can receive a sequence of responses from a fetch call. The Unified API provides a default implementation that logs the callback.

FetchContextStream

Implement this interface to provide a callback that can receive a sequence of responses from a fetch call. An object can be associated with this callback that provides the context of the call. The Unified API provides a default implementation that logs the callback.

TopicDetailsCallback

Implement this interface to provide a callback that receives replies from requests for topic details. The Unified API provides a default implementation that logs the callback.

TopicDetailsContextCallback

Implement this interface to provide a callback that receives replies from requests for topic details. An object can be associated with this callback that provides the context of the call. The Unified API provides a default implementation that logs the callback.

TopicStream

Implement this interface to create a stream that receives notifications of topic events. For more information, see Topic basics on page 127.

DEPRECATED:
Listener

This capability is replaced by `TopicStream`.

Implement this interface to create a listener that receives notifications of topic events. Topic listeners behave differently to standard listeners. They are called in the order that they are registered. A topic listener can consume events it receives and prevent other topic listeners from receiving the event. However, the topic listener mechanism and the topic stream mechanism are independent and a topic listener cannot prevent topic streams from receiving topic events. For more information, see Topic basics on page 127.

Example: Subscribing to a topic

The following examples use the Unified API to subscribe to topics and assign handlers to topics to receive the topic content.

Java

```
package com.pushtechnology.diffusion.examples;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.pushtechnology.diffusion.client.Diffusion;
import com.pushtechnology.diffusion.client.content.Content;
import com.pushtechnology.diffusion.client.content.RecordContentReader;
import com.pushtechnology.diffusion.client.features.Topics;
import com.pushtechnology.diffusion.client.features.Topics.TopicStream;
import com.pushtechnology.diffusion.client.session.Session;
import com.pushtechnology.diffusion.client.types.UpdateContext;

/**
 * In this simple and commonest case for a client we just subscribe to a
 few
```

```
 * topics and assign handlers for each to receive content.
 *
 * This makes use of the 'Topics' feature only.
 *
 * @author Push Technology Limited
 */
public final class ClientSimpleSubscriber {

    private static final Logger LOG =
        LoggerFactory.getLogger(ClientSimpleSubscriber.class);

    private final Session theSession;

    /**
     * Constructor.
     */
    public ClientSimpleSubscriber() {
        // Create the session
        theSession = Diffusion.sessions().open("dpt://localhost:8081");

        // Use the Topics feature to add a topic stream for
        // Foo and all topics under Bar and request subscription to those
 topics
        final Topics topics = theSession.feature(Topics.class);
        topics.addTopicStream(">Foo", new FooTopicStream());
        topics.addTopicStream(">Bar/", new BarTopicStream());
        topics.subscribe(
            Diffusion.topicSelectors().anyOf("Foo", "Bar//"),
            new Topics.CompletionCallback.Default());

        theSession.start();
    }

    /**
     * Close session.
     */
    public void close() {
        theSession.close();
    }

    /**
     * The topic stream for all messages on the 'Foo' topic.
     */
    private class FooTopicStream extends TopicStream.Default {
        @Override
        public void onTopicUpdate(
            String topic,
            Content content,
            UpdateContext context) {

            LOG.info(content.asString());
        }
    }

    /**
     * The topic stream for all messages on 'Bar' topics.
     */
    private class BarTopicStream extends TopicStream.Default {
        @Override
        public void onTopicUpdate(
            String topic,
            Content content,
            UpdateContext context) {

            // Process the message - one with a record with a variable
 number of
```

```
            // fields followed by two more fields (effectively another
 record
            // but no need to process as such).

            final RecordContentReader reader =
                Diffusion.content().newReader(
                    RecordContentReader.class,
                    content);

            for (String field : reader.nextRecord()) {
                LOG.info("Record 1 Field={}", field);
            }

            LOG.info("Extra Field 1={}", reader.nextField());
            LOG.info("Extra Field 2={}", reader.nextField());
        }
    }
}
```

.NET

```
using System;
using PushTechnology.ClientInterface.Client.Session;
using PushTechnology.ClientInterface.Client.Features;
using PushTechnology.ClientInterface.Client.Types;
using PushTechnology.ClientInterface.Client.Topics;
using PushTechnology.ClientInterface.TopicSelectors;
using PushTechnology.ClientInterface.Client.Factories;
using PushTechnology.DiffusionCore.Logging;

namespace PushTechnology.Examples
{
    // A simple client that starts and uses the Topics feature
    // to subscribe to a topic.
    public class ClientSubscribingToTopics
    {
        private ISession session;
        private ITopics topicsFeature;
        private ITopicsCompletionCallback topicsCallback;
        private ITopicStream topicStream;

        public ClientSubscribingToTopics()
        {

            // Get a topics callback
            // The default callback logs if a subscription has
 successfully occurred.
            topicsCallback = new TopicsCompletionCallback();

            // Create a topic stream
            topicStream = new TopicStream();
        }

        public void Open()
        {
            session = Diffusion.Sessions
                .ConnectionTimeout(1200)
                .InputBufferSize(100000)
                .Open("dpt://localhost:8081");

            session.Start();

            // Use the session to get the feature
            topicsFeature = session.GetTopicsFeature();
        }
```

```
        // Add a topic stream to handle topic events and subscribe to the
 topic selector.
        // topicSelector - The topic selector to listen for topic events
 on and subscribe to
        public void SubscribeAndListenForMessages(string topicSelector)
        {
            // Add the topic stream to the feature listening
            // to the topics defined by the path expression
            topicsFeature.AddTopicStream(topicSelector, topicStream);

            // Subscribe to the topic
            topicsFeature.Subscribe(topicSelector, topicsCallback);
        }

        public void Close()
        {
            session.Close();
        }
    }


    // A simple topic stream that logs out all events received on it.
    class TopicStream : ITopicStream
    {
        public void OnSubscription(string topicPath, ITopicDetails
 details)
        {
            LogService.Info("Subscription to " + topicPath);
        }

        public void OnTopicUpdate(string topicPath,
 ClientInterface.Client.Content.IContent content, IUpdateContext context)
        {
            LogService.Info("Update for" + topicPath);
        }

        public void OnUnsubscription(string topicPath,
 TopicUnsubscribeReason reason)
        {
            LogService.Info("Unubscription from " + topicPath);
        }

        public void OnClose()
        {
            LogService.Info("Topic stream closed");
        }

        public void OnError(ClientInterface.Client.Callbacks.ErrorReason
 errorReason)
        {
            LogService.Info("Topic stream closed with error " +
 errorReason);
        }
    }
}
```

C

```
#include <stdio.h>
#include <unistd.h>

#include "diffusion.h"
#include "args.h"

ARG_OPTS_T arg_opts[] = {
        ARG_OPTS_HELP,
```

```
        {'u', "url", "Diffusion server URL", ARG_OPTIONAL, ARG_HAS_VALUE,
 "dpt://localhost:8081"},
        {'t', "topic_selector", "Topic selector", ARG_REQUIRED,
 ARG_HAS_VALUE, NULL},
        END_OF_ARG_OPTS
};

/*
 * This callback is used when the session state changes, e.g. when a
 session
 * moves from a "connecting" to a "connected" state, or from "connected"
 to
 * "closed".
 */
static void
on_session_state_changed(SESSION_T *session,
 const SESSION_STATE_T old_state,
 const SESSION_STATE_T new_state)
{
        printf("Session state changed from %s (%d) to %s (%d)\n",
               session_state_as_string(old_state), old_state,
               session_state_as_string(new_state), new_state);
}

/*
 * This callback may be invoked if an error occurs while the session is
 * changing from one state to another.
 */
static void
on_session_state_error(SESSION_T *session, const DIFFUSION_ERROR_T *error)
{
        printf("Session state error: %s\n", error->message);
}

/*
 * When a subscribed message is received, this callback is invoked.
 */
static int
on_topic_message(SESSION_T *session, const TOPIC_MESSAGE_T *msg)
{
        printf("Received message for topic %s\n", msg->name);
        printf("Payload: %.*s\n", (int)msg->payload->len, msg->payload-
>data);
 return HANDLER_SUCCESS;
}

/*
 * This callback is fired when Diffusion responds to say that a topic
 * subscription request has been received and processed.
 */
static int
on_subscribe(SESSION_T *session, void *context_data)
{
 printf("on_subscribe\n");
 return HANDLER_SUCCESS;
}

/*
 * This is callback is for when Diffusion response to an unsubscription
 * request to a topic, and only indicates that the request has been
 received.
 */
static int
on_unsubscribe(SESSION_T *session, void *context_data)
{
 printf("on_unsubscribe\n");
 return HANDLER_SUCCESS;
```

```
}

/*
 * Publishers and control clients may choose to subscribe any other client
 to
 * a topic of their choice at any time. We register this callback to
 capture
 * messages from these topics and display them.
 */
static int
on_unexpected_topic_message(SESSION_T *session, const TOPIC_MESSAGE_T
 *msg)
{
 printf("Received a message for a topic we didn't subscribe to (%s)\n",
 msg->name);
        printf("Payload: %.*s\n", (int)msg->payload->len, msg->payload-
>data);
 return HANDLER_SUCCESS;
}

/*
 * We use this callback when Diffusion notifies us that we've been
 subscribed
 * to a topic. Note that this could be called for topics that we haven't
 * explicitly subscribed to - other control clients or publishers may ask
 to
 * subscribe us to a topic.
 */
static int
on_notify_subscription(SESSION_T *session, const
 SVC_NOTIFY_SUBSCRIPTION_REQUEST_T *request)
{
 printf("on_notify_subscription: %d: \"%s\"\n",
  request->topic_info.topic_id,
  request->topic_info.topic_name);
 return HANDLER_SUCCESS;
}

/*
 * This callback is used when we receive notification that this client has
 been
 * unsubscribed from a specific topic. Causes of the unsubscription are
 the same
 * as those for subscription.
 */
static int
on_notify_unsubscription(SESSION_T *session, const
 SVC_NOTIFY_UNSUBSCRIPTION_REQUEST_T *request)
{
 printf("on_notify_unsubscription: %d, reason: %d\n",
  request->topic_id,
  request->reason);
 return HANDLER_SUCCESS;
}

int
main(int argc, char **argv)
{
        // Standard command line parsing
        HASH_T *options = parse_cmdline(argc, argv, arg_opts);
        if(options == NULL || hash_get(options, "help") != NULL) {
                show_usage(argc, argv, arg_opts);
                return 1;
        }

        char *url = hash_get(options, "url");
        char *topic = hash_get(options, "topic_selector");
```

```
        // A SESSION_LISTENER_T holds callbacks to inform the client
        // about changes to the state. Used here for informational
        // purposes only.
        SESSION_LISTENER_T session_listener;
        session_listener.on_state_changed = &on_session_state_changed;
        session_listener.on_state_error = &on_session_state_error;

        // Creating a session requires at least a URL. Creating a session
        // initiates a connection with Diffusion and it's possible to send
        // commands, but no topic messages are received until the session
        // is started (session_start).
        DIFFUSION_ERROR_T error;
        SESSION_T *session = NULL;
        session = session_create(url, NULL, NULL, &session_listener, NULL,
&error);
        if(session == NULL) {
                fprintf(stderr, "TEST: Failed to create session\n");
                fprintf(stderr, "ERR : %s\n", error.message);
                return 1;
        }

        // When issuing commands to Diffusion (in this case, subscribe to
        // a topic), it's typical that more than one message may be
        // received in response and a handler can be installed for each
        // message type. In the case of subscription, we can install
        // handlers for:
        // 1. The topic message data (on_topic_message).
        // 2. Notification that the subscription has been received
(on_subscribe).
        // 3. Topic details (on_topic_details).
SUBSCRIPTION_HANDLERS_T *subscription_handlers = calloc(1,
sizeof(SUBSCRIPTION_HANDLERS_T));
subscription_handlers->on_topic_message = &on_topic_message;
subscription_handlers->on_subscribe = &on_subscribe;

UNSUBSCRIPTION_HANDLERS_T *unsubscription_handlers = calloc(1,
sizeof(UNSUBSCRIPTION_HANDLERS_T));
unsubscription_handlers->on_unsubscribe = &on_unsubscribe;

NOTIFY_SUBSCRIPTION_HANDLERS_T *notify_subscription_handlers = calloc(1,
sizeof(NOTIFY_SUBSCRIPTION_HANDLERS_T));
notify_subscription_handlers->on_notify_subscription =
&on_notify_subscription;
svc_notify_subscription_register(session, notify_subscription_handlers);

NOTIFY_UNSUBSCRIPTION_HANDLERS_T *notify_unsubscription_handlers =
calloc(1, sizeof(NOTIFY_UNSUBSCRIPTION_HANDLERS_T));
notify_unsubscription_handlers->on_notify_unsubscription =
&on_notify_unsubscription;
svc_notify_unsubscription_register(session,
notify_unsubscription_handlers);

subscribe(session, topic, subscription_handlers);

// Install a global topic handler to capture messages for topics we
haven't
// explicitly subscribed to, and therefore don't have a specific handler
for.
session->global_topic_handler = on_unexpected_topic_message;

        // Start the session and begin receiving topic messages.
        if(! session_start(session, &error)) {
                fprintf(stderr, "ERR: Failed to start session: %s\n",
error.message);
                return 1;
        }
```

```
        // Keep receiving messages for 5 seconds.
        sleep(5);

 // Unsubscribe from the topic
 unsubscribe(session, topic, unsubscription_handlers);

 // Wait for any unsubscription notifications to be received.
 sleep(5);

        // Politely tell Diffusion we're closing down.
        session_close(session, &error);

        return 0;
}
```

Example: Fetching topic state

The following examples use the Unified API to fetch the current state of a topic without subscribing to the topic.

Java

```
package com.pushtechnology.diffusion.examples;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.pushtechnology.diffusion.client.Diffusion;
import com.pushtechnology.diffusion.client.content.Content;
import com.pushtechnology.diffusion.client.features.Topics;
import
 com.pushtechnology.diffusion.client.features.Topics.FetchContextStream;
import com.pushtechnology.diffusion.client.session.Session;

/**
 * This is a simple example of a client that fetches the state of topics
 but
 * does not subscribe to them.
 *
 * This makes use of the 'Topics' feature only.
 *
 * @author Push Technology Limited
 */
public final class ClientUsingFetch {

    private final Session theSession;
    private final Topics theTopics;

    /**
     * Constructor.
     */
    public ClientUsingFetch() {

        // Create the session
        theSession = Diffusion.sessions().open("dpt://localhost:8081");

        // Get the Topics feature
        theTopics = theSession.feature(Topics.class);

        theSession.start();

    }

    /**
     * Issues a fetch request for a topic or selection of topics.
     *
```

```
     * @param topics a {@link TopicSelector} expression.
     * @param fetchContext context string to be returned with the fetch
 response
     */
    public void fetch(String topics, String fetchContext) {
        theTopics.fetch(topics, fetchContext, new FetchLogger());
    }

    /**
     * Close the session.
     */
    public void close() {
        theSession.close();
    }

    /**
     * A fetch stream handler that simply logs all the topic fetches and
 the end
     * of the fetch stream.
     */
    private static final class FetchLogger implements
        FetchContextStream<String> {

        private static final Logger LOG =
            LoggerFactory.getLogger(FetchLogger.class);

        @Override
        public void onFetchReply(
            String context,
            String topicPath,
            Content content) {
            LOG.info("{} - Fetched {} : {}", context, topicPath,
                content.asString());

        }

        @Override
        public void onClose(String context) {
            LOG.info("{} - Fetch Complete", context);

        }

        @Override
        public void onDiscard(String context) {
            LOG.warn("{} - Fetch Discarded", context);

        }

    }
}
```

.NET

```
using PushTechnology.ClientInterface.Client.Factories;
using PushTechnology.ClientInterface.Client.Features;
using PushTechnology.ClientInterface.Client.Session;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace PushTechnology.Examples
{
    class ClientUsingFetch
    {
        private IFetchContextStream<string> fetchCallback;
```

```
        private ISession session;
        private ITopics topics;

        public ClientUsingFetch()
        {
            fetchCallback = new FetchLogger();
        }

        public void Open()
        {
            session = Diffusion.Sessions.Open("dpt://localhost:8081");
            topics = session.GetTopicsFeature();
            session.Start();
        }

        public void Close()
        {
            session.Close();
        }

        public void Fetch(string topicSelector, string context)
        {
            topics.Fetch(topicSelector, context, fetchCallback);
        }
    }

    class FetchLogger : IFetchContextStream<string>
    {
        public void OnFetchReply(string context, string topicPath,
 ClientInterface.Client.Content.IContent content)
        {
            System.Console.WriteLine("{0} - Fetched {1} : {2}", context,
 topicPath, content.AsString());
        }

        public void OnClose(string context)
        {
            System.Console.WriteLine("{0} - Fetch Complete", context);
        }

        public void OnDiscard(string context)
        {
            System.Console.WriteLine("{0} - Fetch Discarded", context);
        }
    }
}
```

C

```
#include <stdio.h>
#include <unistd.h>

#include "diffusion.h"
#include "args.h"

extern void topic_message_debug();

ARG_OPTS_T arg_opts[] = {
        ARG_OPTS_HELP,
        {'u', "url", "Diffusion server URL", ARG_OPTIONAL, ARG_HAS_VALUE,
 "dpt://localhost:8081"},
        {'t', "topic_selector", "Topic selector", ARG_REQUIRED,
 ARG_HAS_VALUE, NULL},
        {'r', "retries", "Number of connection retries", ARG_OPTIONAL,
 ARG_HAS_VALUE, "3"},
```

```
        {'d', "retry_delay", "Delay (in ms) between connection attempts",
 ARG_OPTIONAL, ARG_HAS_VALUE, "1000"},
 {'p', "principal", "Principal (username) for the connection",
 ARG_OPTIONAL, ARG_HAS_VALUE, NULL},
 {'c', "credentials", "Credentials (password) for the connection",
 ARG_OPTIONAL, ARG_HAS_VALUE, NULL},
        END_OF_ARG_OPTS
};

/**
 * This callback is used when the session state changes, e.g. when a
 session
 * moves from a "connecting" to a "connected" state, or from "connected"
 to
 * "closed".
 */
static void
on_session_state_changed(SESSION_T *session, const SESSION_STATE_T
 old_state, const SESSION_STATE_T new_state)
{
        printf("Session state changed from %s (%d) to %s (%d)\n",
               session_state_as_string(old_state), old_state,
               session_state_as_string(new_state), new_state);
        if(new_state == CONNECTED_ACTIVE) {
                printf("Session ID=%s\n", session_id_to_string(session-
>id));
        }
}

/**
 * This callback may be invoked if an error occurs while the session is
 * changing from one state to another.
 */
static void
on_session_state_error(SESSION_T *session, const DIFFUSION_ERROR_T *error)
{
        printf("Session state error: %s\n", error->message);
}

/**
 * This callback is invoked when Diffusion acknowledges that it has
 received
 * the fetch request. It does not indicate that there will be any
 subsequent
 * messages; see on_topic_message() and on_fetch_status_message() for
 that.
 */
static int
on_fetch(SESSION_T *session, void *context_data)
{
 puts("Fetch acknowledged by server");
 return HANDLER_SUCCESS;
}

/**
 * This callback is invoked when all messages for a topic selector have
 * been received, or there was some kind of server-side error during the
 * fetch processing.
 */
static int
on_fetch_status_message(SESSION_T *session,
 const SVC_FETCH_STATUS_RESPONSE_T *status)
{
 switch(status->status_flag) {
 case DIFFUSION_TRUE:
  puts("Fetch succeeded");
  break; //exit(0);
```

```
 case DIFFUSION_FALSE:
  puts("Fetch failed");
  break; //exit(1);
 default:
  printf("Unknown fetch status: %d\n", status->status_flag);
  break;
 }

 return HANDLER_SUCCESS;
}

/**
 * When a fetched message is received, this callback in invoked.
 */
static int
on_topic_message(SESSION_T *session, const TOPIC_MESSAGE_T *msg)
{
 printf("Received message for topic %s\n", msg->name);
        printf("Payload: %.*s\n", (int)msg->payload->len, msg->payload-
>data);

#ifdef DEBUG
         topic_message_debug(response->payload);
#endif

 return HANDLER_SUCCESS;
}

int
main(int argc, char **argv)
{
 // Standard command line parsing.
        HASH_T *options = parse_cmdline(argc, argv, arg_opts);
        if(options == NULL || hash_get(options, "help") != NULL) {
                show_usage(argc, argv, arg_opts);
                return 1;
        }

        char *url = hash_get(options, "url");
        char *topic = hash_get(options, "topic_selector");
        int retries = atoi(hash_get(options, "retries"));
        long retry_delay = atol(hash_get(options, "retry_delay"));

 // A SESSION_LISTENER_T holds callbacks to inform the client
        // about changes to the state. Used here for informational
        // purposes only.
        SESSION_LISTENER_T foo_listener;
        foo_listener.on_state_changed = &on_session_state_changed;
        foo_listener.on_state_error = &on_session_state_error;

 // The client-side API can automatically keep retrying to connect
 // to the Diffusion server if it's not immediately available.
        SESSION_FAILOVER_STRATEGY_T failover_strategy;
        failover_strategy.retry_count = retries;
        failover_strategy.retry_delay = retry_delay;

 // Creating a session requires at least a URL. Creating a session
        // initiates a connection with Diffusion and it's possible to send
        // commands, but no topic messages are received until the session
        // is started (session_start).
        SESSION_T *session;
        DIFFUSION_ERROR_T error;
        session = session_create(url,
  hash_get(options, "principal"),
  credentials_create_password(hash_get(options, "credentials")),
  &foo_listener, &failover_strategy, &error);
        if(session == NULL) {
```

```
                fprintf(stderr, "TEST: Failed to create session\n");
                fprintf(stderr, "ERR : %s\n", error.message);
                return 1;
        }

        if(! session_start(session, &error)) {
                fprintf(stderr, "ERR: Failed to start session: %s\n",
 error.message);
                return 1;
        }

 // Register handlers for callbacks we're interested in relating to
 // the fetch request. In particular, we want to know about the topic
 // messages that are returned, and the status message which tells
 // us when all messages have been received for the selector (or, if
 // something went wrong.)
 FETCH_HANDLERS_T *handlers = calloc(1, sizeof(FETCH_HANDLERS_T));
 handlers->on_topic_message = on_topic_message;
 handlers->on_fetch = on_fetch;
 handlers->on_status_message = on_fetch_status_message;

 // Issue the fetch request.
 fetch(session, topic, handlers);

 // Wait for up to 5 seconds for the results to come in.
        sleep(1);

 // Clean up politely.
        session_close(session, &error);

        return 0;
}
```

TopicUpdateControl

Use the TopicUpdateControl feature to enable a control client session to update topics.

The TopicUpdateControl feature is available in the following APIs:

Feature	Java	.NET	C
TopicUpdateControl	YES	YES	PARTIAL Does not support paged topics

The TopicUpdateControl feature contains the following classes and interfaces:

`UpdateSource`

Implement this interface to register as an update source for part of the topic tree. The Unified API provides a default implementation.

`Updater`

Receive an object from a callback that implements this interface from the server and use it to update topics in part of the topic tree.

`Updater.Callback`

Implement this interface to provide a callback that receives success or failure responses from update topic calls. The Unified API provides a default implementation that logs the callback.

`Updater.ContextCallback`
Implement this interface to provide a callback that receives success or failure responses from update topic calls. An object can be associated with this callback that provides the context of the call. The Unified API provides a default implementation that logs the callback.

`Updater.ErrorDetails`
This enum lists the reasons that an update action can cause an error.

The TopicUpdateControl feature also contains the following deprecated classes and interfaces:

DEPRECATED:
`TopicSource`
Implement this interface to register as an update source for part of the topic tree. The Unified API provides a default implementation.

DEPRECATED:
`TopicSource.Updater`
Receive an object from a callback that implements this interface from the server and use it to update topics in part of the topic tree.

DEPRECATED:
`TopicSource.Updater.Callback`
Implement this interface to provide a callback that receives success or failure responses from update topic calls. The Unified API provides a default implementation that logs the callback.

DEPRECATED:
`TopicSource.Updater.ContextCallback`
Implement this interface to provide a callback that receives success or failure responses from update topic calls. An object can be associated with this callback that provides the context of the call. The Unified API provides a default implementation that logs the callback.

DEPRECATED:
`TopicSource.Updater.ErrorDetails`
This enum lists the reasons that an update action can cause an error.

Related Links

Building updates for paged topics on page 235
Use the `UpdateFactory` classes with the TopicUpdateControl feature to create the `Update` objects that can be used to update paged topics.

Updating topics from a control client on page 234
A control client can use the TopicUpdateControl feature of the Unified API to update topics.

Control client on page 226
A Diffusion client application can take on the role of control client.

Example: Updating a topic

The following examples use the Unified API to register as the source of a topic and to update that topic with content.

Java

```
package com.pushtechnology.diffusion.examples;

import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.TimeUnit;

import com.pushtechnology.diffusion.client.Diffusion;
import
 com.pushtechnology.diffusion.client.features.control.topics.TopicControl;
```

```
import
 com.pushtechnology.diffusion.client.features.control.topics.TopicControl.AddCallback
import
 com.pushtechnology.diffusion.client.features.control.topics.TopicControl.RemoveCallb
import
 com.pushtechnology.diffusion.client.features.control.topics.TopicUpdateControl;
import
 com.pushtechnology.diffusion.client.features.control.topics.TopicUpdateControl.Updat
import
 com.pushtechnology.diffusion.client.features.control.topics.TopicUpdateControl.Updat
import
 com.pushtechnology.diffusion.client.features.control.topics.TopicUpdateControl.Updat
import com.pushtechnology.diffusion.client.session.Session;
import
 com.pushtechnology.diffusion.client.topics.details.SingleValueTopicDetails;
import com.pushtechnology.diffusion.client.topics.details.TopicDetails;

/**
 * An example of using a control client as an event feed to a topic.
 *
 * This uses the 'TopicControl' feature to create a topic and the
 * 'TopicUpdateControl' feature to send updates to it.
 *
 * @author Push Technology Limited
 */
public class ControlClientAsUpdateSource {

    private static final String TOPIC_NAME = "Feeder";

    private final Session theSession;
    private final TopicControl theTopicControl;
    private final TopicUpdateControl theUpdateControl;
    private final UpdateCallback theUpdateCallback;

    /**
     * Constructor.
     */
    public ControlClientAsUpdateSource() {

        theUpdateCallback = new UpdateCallback.Default();

        // Create the session
        theSession = Diffusion.sessions().open("dpt://localhost:8081");
        theTopicControl = theSession.feature(TopicControl.class);
        theUpdateControl = theSession.feature(TopicUpdateControl.class);

        theSession.start();
    }

    /**
     * Start the feed.
     *
     * @param provider the provider of prices
     * @param scheduler a scheduler service to schedule a periodic feeder
 task
     */
    public void start(
        final PriceProvider provider,
        final ScheduledExecutorService scheduler) {

        // Set up topic details
        final SingleValueTopicDetails.Builder builder =
            theTopicControl
                .newDetailsBuilder(SingleValueTopicDetails.Builder.class);
        final TopicDetails details =

 builder.metadata(Diffusion.metadata().decimal("Price")).build();
```

```
        // Declare a custom update source implementation. When the source
is set
        // as active start a periodic task to poll the provider every
second and
        // update the topic. When the source is closed, stop the scheduled
task.
        final UpdateSource source = new UpdateSource.Default() {
            private ScheduledFuture<?> theFeeder;

            @Override
            public void onActive(String topicPath, Updater updater) {
                theFeeder =
                    scheduler.scheduleAtFixedRate(
                        new FeederTask(provider, updater),
                        1, 1, TimeUnit.SECONDS);
            }

            @Override
            public void onClose(String topicPath) {
                if (theFeeder != null) {
                    theFeeder.cancel(true);
                }
            }
        };

        // Create the topic. When the callback indicates that the topic
has been
        // created then register the topic source for the topic.
        theTopicControl.addTopic(
            TOPIC_NAME,
            details,
            new AddCallback.Default() {
                @Override
                public void onTopicAdded(String topic) {
                    theUpdateControl.registerUpdateSource(topic, source);
                }
            });

    }

    /**
     * Close the session.
     */
    public void close() {
        // Remove our topic and close session when done
        theTopicControl.removeTopics(
            ">" + TOPIC_NAME,
            new RemoveCallback() {
                @Override
                public void onDiscard() {
                    theSession.close();
                }

                @Override
                public void onTopicsRemoved() {
                    theSession.close();
                }
            });

    }

    /**
     * Periodic task to poll from provider and send update to server.
     */
    private final class FeederTask implements Runnable {
```

```
        private final PriceProvider theProvider;
        private final Updater theUpdater;

        private FeederTask(PriceProvider provider, Updater updater) {
            theProvider = provider;
            theUpdater = updater;
        }

        @Override
        public void run() {
            theUpdater.update(
                TOPIC_NAME,
                Diffusion.content().newContent(theProvider.getPrice()),
                theUpdateCallback);
        }

    }

    /**
     * Interface of a price provider that can periodically be polled for a
     * price.
     */
    public interface PriceProvider {
        /**
         * Get the current price.
         *
         * @return current price as a decimal string
         */
        String getPrice();
    }
}
```

.NET

```
using PushTechnology.ClientInterface.Client.Callbacks;
using PushTechnology.ClientInterface.Client.Factories;
using PushTechnology.ClientInterface.Client.Features.Control.Topics;
using PushTechnology.ClientInterface.Client.Session;
using PushTechnology.ClientInterface.Client.Topics;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace PushTechnology.Examples
{
    ///
    /// A control client that uses the ITopicUpdateControl feature to
 register
    /// ITopicUpdateSources and perform updates.
    ///
    class ControlClientAsUpdateSource
    {
        private ISession session;
        private ITopicUpdateControl updateControl;
        private UpdateSource updateSource;
        private ITopicUpdaterUpdateCallback updateCallback;

        public ControlClientAsUpdateSource()
        {
            var registrations = new Dictionary<string, IRegistration>();
            var updateSources = new Dictionary<string, ITopicUpdater>();
            updateSource = new UpdateSource(registrations, updateSources);
            updateCallback = new UpdateCallback();
        }
```

```
        public void Open()
        {
            session = Diffusion.Sessions
                .ConnectionTimeout(1200)
                .InputBufferSize(100000)
                .Open("dpt://localhost:8081");

            updateControl = session.GetTopicUpdateControlFeature();

            session.Start();
        }

        ///
        /// Register an update source against a part of the topic tree.
        ///
        /// topicPath The part of the topic tree
        public void RegisterUpdateSource(string topicPath)
        {
            updateControl.RegisterUpdateSource(topicPath, updateSource);
        }

        ///
        /// Update a single value topic.
        ///
        /// topicPath The topic to update
        /// updateValue The value to update the topic to
        public void UpdateTopic(string topicPath, string updateValue)
        {
            ITopicUpdater updater = updateSource.Updater(topicPath);
            if (updater != null)
            {
                updater.Update(topicPath,
Diffusion.Content.NewContent(updateValue), updateCallback);
            }
            else
            {

updateCallback.OnError(TopicUpdaterErrorReason.INVALID_UPDATER);
            }
        }

        public void Close()
        {
            session.Close();
        }
    }

    ///
    /// The update source to register. It captures the IRegistration and
    /// ITopicUpdater objects to use later.
    ///
    class UpdateSource : TopicUpdateSourceDefault
    {
        private Dictionary<string, IRegistration> registrations;
        private Dictionary<string, ITopicUpdater> updateSources;

        public UpdateSource(Dictionary<string, IRegistration>
registrations, Dictionary<string, ITopicUpdater> updateSources)
        {
            this.registrations = registrations;
            this.updateSources = updateSources;
        }

        public override void OnActive(string topicPath, ITopicUpdater
updater)
        {
            updateSources[topicPath] = updater;
```

```
        }

        public override void OnRegistered(string topicPath, IRegistration
 registration)
        {
            registrations[topicPath] = registration;
        }

        public ITopicUpdater Updater(string topicPath)
        {
            return updateSources[topicPath];
        }
    }

    ///
    /// The callback with the result of the update.
    ///
    class UpdateCallback : ITopicUpdaterUpdateCallback
    {
        public void OnSuccess()
        {
        }

        public void OnError(ErrorReason errorReason)
        {
        }
    }
}
```

C

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

#include <apr.h>
#include <apr_thread_mutex.h>
#include <apr_thread_cond.h>

#include "diffusion.h"
#include "args.h"
#include "conversation.h"
#include "service/svc-update.h"

int active = 0;

apr_pool_t *pool = NULL;
apr_thread_mutex_t *mutex = NULL;
apr_thread_cond_t *cond = NULL;

ARG_OPTS_T arg_opts[] = {
        ARG_OPTS_HELP,
        {'u', "url", "Diffusion server URL", ARG_OPTIONAL, ARG_HAS_VALUE,
 "dpt://localhost:8081"},
 {'t', "topic", "Topic name to create and update", ARG_OPTIONAL,
 ARG_HAS_VALUE, "time"},
        END_OF_ARG_OPTS
};

/*
 * Handlers for add topic feature.
 */
static int
on_topic_added(SVC_ADD_TOPIC_RESPONSE_T *response)
{
```

```
 printf("Added topic\n");
 apr_thread_cond_broadcast(cond);
 return HANDLER_SUCCESS;
}

static int
on_topic_add_failed(SVC_ADD_TOPIC_RESPONSE_T *response)
{
 printf("Failed to add topic (%d)\n", response->response_code);
 apr_thread_cond_broadcast(cond);
 return HANDLER_SUCCESS;
}

static int
on_topic_add_discard(SVC_ADD_TOPIC_RESPONSE_T *response)
{
 apr_thread_cond_broadcast(cond);
 return HANDLER_SUCCESS;
}

/*
 * Handlers for registration of update source feature
 */
static int
on_update_source_init(const CONVERSATION_ID_T *updater_id, const
 SVC_UPDATE_REGISTRATION_RESPONSE_T *response)
{
 printf("Topic source \"%s\" in init state\n",
 conversation_id_to_string(*updater_id));
 apr_thread_cond_broadcast(cond);
 return HANDLER_SUCCESS;
}

static int
on_update_source_registered(const CONVERSATION_ID_T *updater_id, const
 SVC_UPDATE_REGISTRATION_RESPONSE_T *response)
{
 printf("Registered update source \"%s\"\n",
 conversation_id_to_string(*updater_id));
 apr_thread_cond_broadcast(cond);
 return HANDLER_SUCCESS;
}

static int
on_update_source_active(const CONVERSATION_ID_T *updater_id, const
 SVC_UPDATE_REGISTRATION_RESPONSE_T *response)
{
 printf("Topic source \"%s\" active\n",
 conversation_id_to_string(*updater_id));
 active = 1;
 apr_thread_cond_broadcast(cond);
 return HANDLER_SUCCESS;
}

static int
on_update_source_standby(const CONVERSATION_ID_T *updater_id, const
 SVC_UPDATE_REGISTRATION_RESPONSE_T *response)
{
 printf("Topic source \"%s\" on standby\n",
 conversation_id_to_string(*updater_id));
 apr_thread_cond_broadcast(cond);
 return HANDLER_SUCCESS;
}

static int
on_update_source_closed(const CONVERSATION_ID_T *updater_id, const
 SVC_UPDATE_REGISTRATION_RESPONSE_T *response)
```

```
{
 printf("Topic source \"%s\" closed\n",
 conversation_id_to_string(*updater_id));
 apr_thread_cond_broadcast(cond);
 return HANDLER_SUCCESS;
}

/*
 * Handlers for update of data.
 */
static int
on_update_success(const CONVERSATION_ID_T *updater_id, const
 SVC_UPDATE_RESPONSE_T *response)
{
 printf("on_update_success for updater \"%s\"\n",
 conversation_id_to_string(*updater_id));
 return HANDLER_SUCCESS;
}

static int
on_update_failure(const CONVERSATION_ID_T *updater_id, const
 SVC_UPDATE_RESPONSE_T *response)
{
 printf("on_update_failure for updater \"%s\"\n",
 conversation_id_to_string(*updater_id));
 return HANDLER_SUCCESS;
}

/*
 *
 */
int
main(int argc, char** argv)
{
 // Standard command line parsing.
        HASH_T *options = parse_cmdline(argc, argv, arg_opts);
        if(options == NULL || hash_get(options, "help") != NULL) {
                show_usage(argc, argv, arg_opts);
                return 1;
        }

        char *url = hash_get(options, "url");
 char *topic_name = hash_get(options, "topic");

 // Setup for condition variable
 apr_initialize();
 apr_pool_create(&pool, NULL);
 apr_thread_mutex_create(&mutex, APR_THREAD_MUTEX_DEFAULT, pool);
 apr_thread_cond_create(&cond, pool);

 // Setup for session
 SESSION_T *session;
 DIFFUSION_ERROR_T error;
 session = session_create(url, NULL, NULL, NULL, NULL, &error);
 if(session == NULL) {
                fprintf(stderr, "TEST: Failed to create session\n");
                fprintf(stderr, "ERR : %s\n", error.message);
                return 1;
        }
        if(! session_start(session, &error)) {
                fprintf(stderr, "ERR: Failed to start session: %s\n",
 error.message);
                return 1;
        }

 // Handlers for add_topic()
```

```
 ADD_TOPIC_HANDLERS_T *add_topic_handlers = calloc(1,
 sizeof(ADD_TOPIC_HANDLERS_T));
 add_topic_handlers->on_topic_added = on_topic_added;
 add_topic_handlers->on_topic_add_failed = on_topic_add_failed;
 add_topic_handlers->on_discard = on_topic_add_discard;

 // Single value string data, no default data.
 TOPIC_DETAILS_T *string_topic_details =
 create_topic_details_single_value(M_DATA_TYPE_STRING);
 apr_thread_mutex_lock(mutex);
 add_topic(session, topic_name, string_topic_details, NULL,
 add_topic_handlers);
 apr_thread_cond_wait(cond, mutex);
 apr_thread_mutex_unlock(mutex);

 // Handlers for add_update_source()
 UPDATE_SOURCE_REGISTRATION_HANDLERS_T
 *update_source_registration_handlers = calloc(1,
 sizeof(UPDATE_SOURCE_REGISTRATION_HANDLERS_T));
 update_source_registration_handlers->on_init = on_update_source_init;
 update_source_registration_handlers->on_registered =
 on_update_source_registered;
 update_source_registration_handlers->on_active = on_update_source_active;
 update_source_registration_handlers->on_standby =
 on_update_source_standby;
 update_source_registration_handlers->on_close = on_update_source_closed;

 // Register an updater
 apr_thread_mutex_lock(mutex);
 CONVERSATION_ID_T *updater_id = register_update_source(session,
 topic_name, update_source_registration_handlers);
 apr_thread_cond_wait(cond, mutex);
 apr_thread_mutex_unlock(mutex);

 while(active) {
  BUF_T *buf = buf_create();
  time_t time_now = time(NULL);
  buf_write_string(buf, ctime(&time_now));

  CONTENT_T *content =
   content_create(CONTENT_ENCODING_NONE, buf);

  UPDATE_T *upd =
   update_create(UPDATE_ACTION_MATCH,
     UPDATE_TYPE_CONTENT,
     content);

  UPDATE_SOURCE_HANDLERS_T *update_handlers = calloc(1,
 sizeof(UPDATE_SOURCE_HANDLERS_T));
  update_handlers->on_success = on_update_success;
  update_handlers->on_failure = on_update_failure;

  update(session, updater_id, topic_name, upd, update_handlers);

  sleep(1);
  update(session, updater_id, topic_name, upd, update_handlers);
 }


 sleep(10000);

 return 0;
}
```

Related Links

Control client on page 226

A Diffusion client application can take on the role of control client.

Example: Updating a paged topic

The following examples use the Unified API to create a paged topic and to update that paged topic with content.

Java

```
package com.pushtechnology.diffusion.examples;

import com.pushtechnology.diffusion.client.Diffusion;
import com.pushtechnology.diffusion.client.content.metadata.MRecord;
import
 com.pushtechnology.diffusion.client.content.metadata.MetadataFactory;
import
 com.pushtechnology.diffusion.client.content.update.PagedRecordOrderedUpdateFactory;
import
 com.pushtechnology.diffusion.client.content.update.PagedStringUnorderedUpdateFactory
import com.pushtechnology.diffusion.client.content.update.Update;
import
 com.pushtechnology.diffusion.client.features.control.topics.TopicControl;
import
 com.pushtechnology.diffusion.client.features.control.topics.TopicControl.AddCallback
import
 com.pushtechnology.diffusion.client.features.control.topics.TopicControl.RemoveCallb
import
 com.pushtechnology.diffusion.client.features.control.topics.TopicUpdateControl;
import
 com.pushtechnology.diffusion.client.features.control.topics.TopicUpdateControl.Updat
import
 com.pushtechnology.diffusion.client.features.control.topics.TopicUpdateControl.Updat
import
 com.pushtechnology.diffusion.client.features.control.topics.TopicUpdateControl.Updat
import com.pushtechnology.diffusion.client.session.Session;
import
 com.pushtechnology.diffusion.client.topics.details.PagedRecordTopicDetails;
import
 com.pushtechnology.diffusion.client.topics.details.PagedRecordTopicDetails.Attribute
import com.pushtechnology.diffusion.client.topics.details.TopicType;

/**
 * An example of using a control client to create and update paged topics.
 *
 * This uses the 'TopicControl' feature to create a paged topic and the
 * 'TopicUpdateControl' feature to send updates to it.
 *
 * This demonstrates some simple examples of paged topic updates but not
 all of
 * the possible ways in which they can be done.
 *
 * @author Push Technology Limited
 */
public class ControlClientUpdatingPagedTopics {

    private static final String ORDERED_TOPIC = "Paged/Ordered";
    private static final String UNORDERED_TOPIC = "Paged/Unordered";

    private final Session theSession;
    private final TopicControl theTopicControl;
    private final TopicUpdateControl theUpdateControl;
    private final UpdateCallback theUpdateCallback;
    private final PagedRecordOrderedUpdateFactory theOrderedUpdateFactory;
    private final PagedStringUnorderedUpdateFactory
 theUnorderedUpdateFactory;
    private Updater theUpdater = null;
```

```
    /**
     * Constructor.
     */
    public ControlClientUpdatingPagedTopics() {

        theUpdateCallback = new UpdateCallback.Default();

        // Create the session
        theSession = Diffusion.sessions().open("dpt://localhost:8081");
        theTopicControl = theSession.feature(TopicControl.class);
        theUpdateControl = theSession.feature(TopicUpdateControl.class);

        theOrderedUpdateFactory =
            theUpdateControl
                .updateFactory(PagedRecordOrderedUpdateFactory.class);
        theUnorderedUpdateFactory =
            theUpdateControl
                .updateFactory(PagedStringUnorderedUpdateFactory.class);

        theSession.start();

        final MetadataFactory metadata = Diffusion.metadata();

        // Create an unordered paged string topic
        theTopicControl.addTopic(
            UNORDERED_TOPIC,
            theTopicControl.newDetails(TopicType.PAGED_STRING),
            new AddCallback.Default());

        // Create an ordered paged record topic
        final MRecord recordMetadata =
            metadata.record(
                "Record",
                metadata.string("Name"),
                metadata.string("Address"));

        theTopicControl.addTopic(
            ORDERED_TOPIC,
            theTopicControl.
                newDetailsBuilder(PagedRecordTopicDetails.Builder.class).
                metadata(recordMetadata).
                order(new OrderKey("Name")).build(),
            new AddCallback.Default());

        // Register as updater for topics under the 'Paged' branch
        theUpdateControl.registerUpdateSource(
            "Paged",
            new UpdateSource.Default() {
                @Override
                public void onActive(String topicPath, Updater updater) {
                    theUpdater = updater;
                }
            });
    }

    /**
     * Add a new line to the ordered topic.
     *
     * @param name the name field value
     * @param address the address field value
     */
    public void addOrdered(String name, String address) {
        update(
            ORDERED_TOPIC,
            theOrderedUpdateFactory.add(
                Diffusion.content().newRecord(name, address)));
    }
```

```
    /**
     * Update a line of an ordered topic.
     *
     * @param name the name of the line to update
     * @param address the new address field value
     */
    public void updateOrdered(String name, String address) {
        update(
            ORDERED_TOPIC,
            theOrderedUpdateFactory.update(
                Diffusion.content().newRecord(name, address)));
    }

    /**
     * Remove a line from an ordered topic.
     *
     * @param name the name of the line to remove
     */
    public void removeOrdered(String name) {
        update(
            ORDERED_TOPIC,
            theOrderedUpdateFactory.remove(
                Diffusion.content().newRecord(name, "")));
    }

    /**
     * Add a line or lines to the end of an unordered topic.
     *
     * @param values lines to add
     */
    public void addUnordered(String... values) {
        update(
            UNORDERED_TOPIC,
            theUnorderedUpdateFactory.add(values));
    }

    /**
     * Insert a line or lines at a specified index within an unordered
topic.
     *
     * @param index the index to add at
     * @param values lines to insert
     */
    public void insertUnordered(int index, String... values) {
        update(
            UNORDERED_TOPIC,
            theUnorderedUpdateFactory.insert(index, values));
    }

    /**
     * Update a line within an unordered topic.
     *
     * @param index the index of the line to update
     * @param value the new line value
     */
    public void updateUnordered(int index, String value) {
        update(
            UNORDERED_TOPIC,
            theUnorderedUpdateFactory.update(index, value));
    }

    /**
     * Remove a specific line from an unordered topic.
     *
     * @param index the line to remove
     */
```

```
    public void removeUnordered(int index) {
        update(
            UNORDERED_TOPIC,
            theUnorderedUpdateFactory.remove(index));
    }

    private void update(String topic, Update update)
        throws IllegalStateException {
        if (theUpdater == null) {
            throw new IllegalStateException("No updater");
        }
        theUpdater.update(topic, update, theUpdateCallback);
    }

    /**
     * Close the session.
     */
    public void close() {
        // Remove our topic and close session when done
        theTopicControl.removeTopics(
            ">Paged",
            new RemoveCallback() {
                @Override
                public void onDiscard() {
                    theSession.close();
                }

                @Override
                public void onTopicsRemoved() {
                    theSession.close();
                }
            });

    }
}
```

.NET

```
using PushTechnology.ClientInterface.Client.Callbacks;
using PushTechnology.ClientInterface.Client.Factories;
using PushTechnology.ClientInterface.Client.Features.Control.Topics;
using PushTechnology.ClientInterface.Client.Session;
using PushTechnology.ClientInterface.Client.Topics.Update;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace PushTechnology.Examples
{
    ///
    /// A control client that uses the ITopicUpdateControl feature to
 register
    /// ITopicUpdateSources and perform updates on paged topic data. This
    /// relies on the ControlClientAddingPagedTopics to create the paged
    /// topics.
    ///
    class ControlClientAsUpdateSourceForPagedTopics
    {
        private const string ORDERED_TOPIC = "Paged/Ordered";
        private const string UNORDERED_TOPIC = "Paged/Unordered";

        private ISession session;
        private ITopicUpdateControl updateControl;
        private PagedUpdateSource updateSource;
        private ITopicUpdaterUpdateCallback updateCallback;
```

```
        private IPagedRecordOrderedUpdateFactory orderedUpdateFactory;
        private IPagedStringUnorderedUpdateFactory unorderedUpdateFactory;

        public ControlClientAsUpdateSourceForPagedTopics()
        {
            var registrations = new Dictionary<string, IRegistration>();
            var updateSources = new Dictionary<string, ITopicUpdater>();
            updateSource = new PagedUpdateSource();
            updateCallback = new UpdateCallback();
        }

        public void Open()
        {
            session = Diffusion.Sessions
                .ConnectionTimeout(1200)
                .InputBufferSize(100000)
                .Open("dpt://localhost:8081");

            // Use the session to get the feature
            updateControl = session.GetTopicUpdateControlFeature();

            // Use the feature to get update factories
            orderedUpdateFactory =
updateControl.UpdateFactory<IPagedRecordOrderedUpdateFactory>();
            unorderedUpdateFactory =
updateControl.UpdateFactory<IPagedStringUnorderedUpdateFactory>();

            session.Start();

            // Register an update source for all paged topics
            updateControl.RegisterUpdateSource("Paged", updateSource);
        }

        public void Close()
        {
            session.Close();
        }

        ///
        /// Add a new line to the ordered record topic.
        ///
        /// name  The name to add
        /// address   The address to add
        public void AddOrdered(String name, String address)
        {
            Update(ORDERED_TOPIC, orderedUpdateFactory.Add(
                Diffusion.Content.NewRecord(name, address)));
        }

        ///
        /// Update an existing line in the ordered record topic.
        ///
        /// name  The name to find the line (because the topic is ordered
on this field)
        /// address  The address to update
        public void UpdateOrdered(String name, String address)
        {
            Update(ORDERED_TOPIC, orderedUpdateFactory.Update(
                Diffusion.Content.NewRecord(name, address)));
        }

        ///
        /// Remove an existing line in the ordered record topic.
        ///
        /// name   The name of the line to remove
        public void RemoveOrdered(String name)
        {
```

```
            Update(ORDERED_TOPIC, orderedUpdateFactory.Remove(
                Diffusion.Content.NewRecord(name, "")));
        }

        ///
        /// Add a new line to the unordered string topic.
        ///
        /// values   The fields to add as a line
        public void AddUnordered(params string[] values)
        {
            Update(UNORDERED_TOPIC, unorderedUpdateFactory.Add(values));
        }

        ///
        /// Insert a new line to the unordered string topic at a given
index.
        ///
        /// index  The index to insert at
        /// values  The fields to insert as a line
        public void InsertUnordered(int index, params string[] values)
        {
            Update(UNORDERED_TOPIC, unorderedUpdateFactory.Insert(index,
values));
        }

        ///
        /// Update a line of the unordered string topic at a given index.
        ///
        /// index
        /// value
        public void UpdateUnordered(int index, string value)
        {
            Update(UNORDERED_TOPIC, unorderedUpdateFactory.Update(index,
value));
        }

        ///
        /// Remove a line of the unordered string topic at a given index.
        ///
        /// index  The index of the line to remove
        public void RemoveUnordered(int index)
        {
            Update(UNORDERED_TOPIC, unorderedUpdateFactory.Remove(index));
        }

        private void Update(string topic, IUpdate update)
        {
            ITopicUpdater updater = updateSource.Updater("Paged");
            if (updater != null)
            {
                // Use the updater to update the topic
                updater.Update(topic, update, updateCallback);
            }
        }
    }

    ///
    /// Update source that captures the Updater.
    ///
    class PagedUpdateSource : TopicUpdateSourceDefault
    {
        private Dictionary<string, ITopicUpdater> updateSources;

        public PagedUpdateSource()
        {
            this.updateSources = new Dictionary<string, ITopicUpdater>();
        }
```

```
        public override void OnActive(string topicPath, ITopicUpdater
 updater)
        {
            updateSources[topicPath] = updater;
        }

        public ITopicUpdater Updater(string topicPath)
        {
            return updateSources[topicPath];
        }
    }
}
```

Related Links

Control client on page 226
A Diffusion client application can take on the role of control client.

Chapter 12

Classic APIs

In this section:

- Table of APIs
- Java API
- .NET API
- JavaScript API
- ActionScript API
- Silverlight API
- iOS API
- Android API
- C API
- diffusion-wrapper.js

Diffusion provides a number of Application Programming Interfaces (APIs) which allow user-written applications to make use of Diffusion.

The APIs documented in this section are still supported for version 5.1, but in future is replaced by the new API. For more information, see Unified API on page 240

Where an API is not available for a particular language a user can still communicate with Diffusion through a TCP socket based connection using the Diffusion Protocol.

Related Links

Diffusion APIs on page 42
Diffusion provides application programming interfaces (APIs) that you can use to create applications that interact with the Diffusion server.

Unified API on page 240
The Diffusion Unified application programming interface (API) provides a consistent interface to the Diffusion server, whichever role your client application performs: standard client or control client.

Table of APIs

Diffusion provides APIs in a variety of languages. Each of these APIs is supported over a set of protocols and in a specific level of the language.

	Publisher	Event publisher	Client	Implementation version	Protocols
Java	YES	YES	YES	Java 7 or Java 8	• DPT, DPTS • WS, WSS • HTTP, HTTPS (Full duplex)
.NET		YES	YES	.NET 3.5	• DPT, DPTS • WS, WSS • HTTP (Full duplex)
JavaScript			YES	JavaScript V1.3	• Native: WS, WSS, HTTP, HTTPS, HTTP Streaming • Flash: DPT, HTTP, HTTPS, HTTP Streaming • Silverlight: DPT, HTTP, HTTPS
Flash			YES	ActionScript V3.0 (Flex 3.0)	• DPT, DPTS • HTTP, HTTPS • HTTPC, HTTPCS
Silverlight			YES	Silverlight V4.0	• DPT • HTTP, HTTPS • HTTPC, HTTPCS
iOS			YES	iOS v5.1.1, architectures: armv7, armv7s, arm64	• DPT, DPTS • WS, WSS
Android			YES	Android 2.2	• DPT, DPTS

Tier 2 APIs supported on a best effort basis

	Publisher	Event publisher	Client	Implementatio version	>Protocols
C			YES		• DPT

For more information on the transports supported by the clients please refer to the section Transports supported

Table 75: Feature matrix

Feature	Java	.NET	JavaScript	C	Flex	Silverlight	Java ME	iOS	Android
Asynchronous connect		YES	YES		YES	YES		YES	
Simple connect	YES	YES	YES	YES	YES	YES	YES	YES	YES
Connect with topics (variable string args)	YES	YES	YES						
Connect with topicSet	YES	YES							
Connect with multiple server details	YES	YES		YES	YES	YES	YES	YES	YES
Connection cascading	YES	YES		YES	YES	YES	YES	YES	YES
Connection failover	YES	YES			YES	YES	YES	YES	YES
Connection load balancing	YES	YES		YES	YES	YES	YES	YES	YES
Reconnect	YES	YES	YES	YES	YES	YES	YES	YES	YES
Get client ID	YES	YES	YES	YES	YES	YES	YES	YES	YES
Is Connected status	YES	YES	YES		YES	YES	YES	YES	YES
Is Reconnected status	YES	YES	YES			YES	YES	YES	YES
Ping server	YES	YES	YES	YES	YES	YES	YES	YES	YES
Create topic load message object	YES	YES					YES		YES
Create delta message object	YES	YES	YES	YES	YES	YES	YES	YES	YES
Send (topic load) message	YES	YES	YES				YES		YES
Send (delta) message	YES	YES	YES	YES	YES	YES	YES	YES	YES
(Auto) message acknowledgment	YES	YES	YES	YES	YES	YES	YES	YES	YES
(Manual) message acknowledgment	YES	YES	YES	YES	YES	YES	YES	YES	YES
Subscribe with topics (variable string args)	YES	YES					YES		YES

Feature	Java	.NET	JavaScript	C	Flex	Silverlight	Java ME	iOS	Android
Subscribe with topicSet	YES	YES							
Fetch with topicSet	YES	YES							
Fetch	YES	YES	YES	YES	YES	YES	YES	YES	YES
Unsubscribe with topics (variable string args)	YES	YES							
Unsubscribe with topic	YES	YES	YES	YES	YES	YES	YES	YES	YES
Unsubscribe with topicSet	YES	YES							
Create delta message	YES	YES							
Create topic load message	YES	YES							
Add topic listener	YES	YES	YES	YES	YES	YES	YES	YES	YES
Remove topic listener	YES	YES	YES	YES	YES	YES	YES	YES	YES
Add Timed topic listener					YES				
Read-only access into topic listeners		YES							
Create service topic handler	YES	YES	YES	YES	YES	YES	YES	YES	YES
Create paged topic handler	YES	YES	YES	YES	YES	YES	YES	YES	YES
Create topic notify topic handler	YES	YES	YES		YES		YES	YES	YES
Fragmented messages	YES	YES	YES	YES	YES	YES	YES	YES	YES

Java API

The Java API comprises a number of packages all subordinate to the main `com.pushtechnology.diffusion.api` package.

Full API documentation is issued with the product. For information about how to use the API components, see the API documentation. The following table is a guide to which packages contain the major API components.

Table 76: Java APIs

API area	Relevant packages
Publishers	The main classes for writing a publisher are in the `publisher` package. General classes relating to topic handling are in `topic`

API area	Relevant packages
	Classes related to messages are in `message`. All classes relating to the use of TopicData are in the `data` package. Classes relating to the use of conflation are in the `conflation` package. See Publishers for more information about how to use the publisher API.
Client	The client API classes are in the `client` package and makes use of the `connection` package for connection details. Client applications also make use of `config` for configuration and `message` for message handling. The `threads` package also applies for thread pool handling.
DEPRECATED: Event Publisher	The event publisher API is located in the `evpub` package and it also uses the same common packages that the client API uses. See Event publishers for more information about using an event publisher.
Server	Classes relating to the Diffusion server (running it within an application) are located in the `server` package and the configuration of the server is done with the classes in the `config` package.
Web server	Classes relating to implementing Diffusion web server functionality are in the `webserver` package.
Configuration	The `config` package contains all configuration classes used for configuring the Diffusion server and/or within client APIs.
System management	System management classes are located in the `management` package.
Statistics	Classes related to statistics are in the `statistics` package.
Whols	If you want to implement your own Whols provider, the classes required are in the `whois` package.
General	General purpose classes relating to exceptions and logging as well as some useful utility classes are located directly under the `api` package itself.

Client API

The client API provides the ability to connect to a Diffusion server as an external client from within any Java application.

How to use the Java client API

There is a single class called `ExternalClientConnection` which can be instantiated with the required connection details and used to make an actual connection.

The connection class is of the generic type `ServerConnection` and as such, once a connection is made, any notifications or messages from the server are passed through the `ServerConnectionListener` interface.

The topic or topics to subscribe to can be specified when connecting or at any time after connection.

The `ServerConnectionListener` specified will receive all messages for all topics. However, any number of additional topic listeners can be specified and messages for different topics routed to different listeners as required.

The API permits the following types of connection to be specified by using the `ServerDetails` (see connection package) specified when configuring the connection object:

Table 77: Connection types

TCP	For a standard connection over TCP/IP. This must connect to a standard client connector.
SSL	For a secure TCP/IP connection over SSL. This must connect to a client connector with SSL enabled
HTTP	For a connection using HTTP
HTTP/SSL	For a connection using HTTP over SSL.

By specifying more than one `ServerDetails`, fallback connections can be specified. If the first connection does not succeed, the second is tried, and so on.

For a detailed description of the API see the issued API documentation (in docs directory).

Authorization credentials

If authorization credentials are required by the Diffusion server, these are set at the `ConnectionDetails` level and used for all `ServerDetails`. Credentials can be set in a `ServerDetails` by creating a `Credentials` object and using `setCredentials` before connecting.

Credentials can also be sent to the server after connection using the method `sendCredentials` in `ExternalClientConnection`. In this case the credentials can be rejected by the server, in which case this is notified on the `serverRejectedCredentials` method of each `ServerConnectionListener`.

Reconnection

If a client unexpectedly loses connection, it can try to reconnect using the `reconnect` method. If the server has specified keep-alive for the connector, the client can pick up the same session as before and receive all messages that were queued for it whilst disconnected. The topic state (that is, which topics the client is subscribed to) is also re-established on reconnection. If unable to reconnect, a new connection is established with the same topic set as used on the original connection. Successful reconnection or connection is notified on the normal `serverConnected` method and you can determine which has occurred using the `ServerConnection.isReconnected()` method.

There is no guarantee that messages in transit at the time of the disconnection will be redelivered. However, all messages marked as requiring acknowledgment by the server are delivered.

Failover

The Java client supports autofailover. For more information, see the Java Failover documentation.

Special features

Paged topic data handling

Where paged topic data is in use at the server there are features within the client API which simplify the handling of messages to and from such a topic.

Service topic data handling

Where service topic data is in use at the server there are features within the client API which simplify the handling of messages to and from such a topic.

Example: Simple client class

The following example shows a simple client class which sends a message containing "Hello" to the server and logs all messages it receives, until it receives a message from the server (Publisher) asking it to stop. It tries to connect through TCP first but if that fails it tries HTTP.

```
public class ClientApplication implements
 ServerConnectionListener {

    private static final Logger LOG =
        LoggerFactory.getLogger(ClientApplication.class);

    private ExternalClientConnection theConnection;

    public ClientApplication() throws APIException {
        // Create Connection
        theConnection=
            new ExternalClientConnection(
                this,
                "dpt://localhost:8080",
                "http://localhost:8080");
        // Connect, subscribing to a single topic
        theConnection.connect("MyTopic");
        // Send a message
        TopicMessage message =
 theConnection.createDeltaMessage("MyTopic");
        message.put("Hello");
        theConnection.send(message);
    }

    public void messageFromServer(
    ServerConnection serverConnection,
    TopicMessage message) {
        LOG.info("Message Received : {}",message);
        try {
            if (message.asString().equals("STOP")) {
                theConnection.close();
            }
        }
        catch(Exception ex) {
            ex.printStackTrace();
        }
    }

    public void serverConnected(ServerConnection
 serverConnection){
        LOG.info("Connected to Server : {}",serverConnection);

    }

    public void serverTopicStatusChanged(
    ServerConnection serverConnection,
    String topicName,
    TopicStatus status) {
        LOG.info(
            "Topic {} at {} status changed to {}",
            topicName,serverConnection,status);
    }

    public void serverRejectedCredentials(
    ServerConnection serverConnection,
    Credentials credentials) {
        LOG.info("Server Rejected Credentials :
 {}",serverConnection);
```

```
    }

    public void serverDisconnected(ServerConnection
 serverConnection) {
        LOG.info("Disconnected from Server :
 {}",serverConnection);
    }

}
```

Related Links

Using Maven to build Java Diffusion applications on page 600
Apache Maven is a popular Java build tool and is well supported by Java IDEs. You can use Apache Maven to build your Diffusion applications.

(Deprecated) Event publisher API

How to use the Java event publisher API

> **Caution:**
>
> The event publisher API is deprecated. Interactions using this API are still supported in Diffusion version 5.1, but this feature is replaced in later versions.
>
> You can use the Unified API to develop a control client that has equivalent capabilities to an event publisher.

The main class is `EventPublisherConnection` which is instantiated to configure the connection and `connect` is called to make an actual connection.

Messages are sent to the server and each is routed to the publisher that owns the message topic. The server can also send messages to the event publisher.

The connection can be made only over DPT and as such `ServerDetails` might not be specified but can be obtained to change details before connection.

The port specified to the connection must correspond to an event publisher connector for a Diffusion server running on the specified host.

For a detailed description of the API see the issued API documentation (in docs directory).

The following example shows the simple case of establishing a connection and sending a single message, though in reality an event publisher sends many messages over a period of time.

```
public class EventPublisherApplication implements EventPublisherListener {

    private static final Logger LOG =
        LoggerFactory.getLogger(EventPublisherApplication.class);

    private EventPublisherConnection theConnection;

    public EventPublisherApplication() throws APIException {
        theConnection=new EventPublisherConnection("localhost",3098,this);
        theConnection.getServerDetails().setOutputBufferSize(8000);
        theConnection.connect();
        TopicMessage message =
 theConnection.createDeltaMessage("MyTopic",0);
        theConnection.send(message);
    }

    public void messageFromServer(
    EventPublisherConnection connection,
    TopicMessage message) {
        LOG.info(
            "Event publisher Connection {} received message from server :
 {}",
            connection,message);
```

```
    }

    public void serverDisconnected(EventPublisherConnection connection) {
        LOG.info(
            "Event publisher Connection Closed {}",connection);
    }

}
```

.NET API

The .NET API comprises a number of packages.

Client API

The ExternalClient API provides the ability to connect to a Diffusion server as an external client from within any .NET application.

There is a single class called `ExternalClient` which can be instantiated with the required connection details and used to make an actual connection.

The topic or topics to subscribe to can be specified when connecting or at any time after connection.

When a connection object is instantiated, subscribe to the `InboundMessageReceived` delegate, which receives all messages for all topics.

The API permits the following types of connection to be specified by using the `ServerDetails` specified when configuring the connection object:

Table 78: Types of connection that can be specified from the .NET client

TCP	For a standard connection over DPT. This connects to the Java external client connector.
SSL	For a secure TCP/IP connection over DPTS. This connects to the Java external client connector.
HTTP	For a connection using HTTP Protocol

For a detailed description of the API, see the issued documentation (in docs directory).

The following example shows a simple client class which sends a message containing 'Hello' to the server until it receives a message from the server (Publisher) asking it to stop.

```
   public class ClientApplication : IServerConnectionListener,
ITopicListener
{
 #region Fields

 private readonly PushTechnology.DiffusionExternalClient.ExternalClient
theClient;

 #endregion // Fields

 #region Constructor

 public ClientApplication()
 {
  var connectionDetails =
   ConnectionFactory.CreateConnectionDetails( "dpt://localhost:8080",
"http://localhost:8080" );
```

```
  connectionDetails.Topics = new TopicSet("MyTopic");

  theClient = new
PushTechnology.DiffusionExternalClient.ExternalClient(connectionDetails);

  // Add a topic listener - we are listening to all messages for this
example, but individual topics can
  // also be used as selectors
  theClient.AddGlobalTopicListener( this );

  // Now connect - this is an asynchronous process, so we have to wait
until ServerConnected is invoked
  theClient.Connect();
 }

 #endregion // Constructor

 #region Implementation of IServerConnectionListener

 /// <summary>
 /// Notification of connection.
 ///
 /// This is called when a connection to a server is established.
 /// </summary>
 /// <param name="connector">The server connector.</param>
 public void ServerConnected( IDiffusionClientConnector connector )
 {
  Console.WriteLine( "Connected to server: " + connector );

  // Send a message as we are now connected
  ITopicMessage message = theClient.CreateDeltaMessage( "MyTopic" );

  // Populate the message
  message.Put( "Hello" );

  // Send the message to the Diffusion server
  theClient.SendMessage( message );
 }

 /// <summary>
 /// Notification that the status for a topic that was subscribed to has
changed.
 /// </summary>
 /// <param name="connector">The connector.</param>
 /// <param name="topicName">The name of the topic on which the status
has changed.</param>
 /// <param name="statusType">The topic status change type.</param>
 public void ServerTopicStatusChanged( IDiffusionClientConnector
connector, string topicName, TopicStatusChangeType statusType )
 {
  Console.WriteLine(
   string.Format( "Topic status for '{0}' changed to '{1}'.", topicName,
statusType ));
 }

 /// <summary>
 /// Notification of rejected credentials from the server.
 /// </summary>
 /// <param name="connector"></param>
 /// <param name="credentials"></param>
 public void ServerRejectedCredentials( IDiffusionClientConnector
connector, Credentials credentials )
 {
  Console.WriteLine( "Server rejected credentials.");
 }

 /// <summary>
```

```
  /// Notification of disconnection.
  ///
  /// The reason for the disconnection can be established by checking the
state of the connection
  /// using IDiffusionClientConnector.State.
  /// </summary>
  /// <param name="connector">The server connector.</param>
  /// <param name="args">The arguments which can be interrogated for the
state and details of a server closure.</param>
  public void ServerDisconnected( IDiffusionClientConnector connector,
ServerClosedEventArgs args )
  {
   Console.WriteLine( "Disconnected from server.");
  }

  #endregion

  #region Implementation of ITopicListener

  /// <summary>
  /// Handles a message received from an IMessageSource.
  ///
  /// This handles an incoming message from a specified source.
  /// </summary>
  /// <param name="source">The message source.</param>
  /// <param name="message">The message.</param>
  public bool HandleTopicMessage( IMessageSource source, ITopicMessage
message )
  {
   if (message.AsString().Equals("STOP"))
   {
    theClient.Disconnect();
   }

   return false;
  }

  #endregion
 }
```

Connection events

Events that are invoked when a connection to the Diffusion server is established, fails, or is lost are invoked synchronously.

The following connection events are invoked synchronously by the .NET client library:

- `DiffusionServerConnected` is invoked when the client library successfully establishes a connection to the Diffusion server
- `DiffusionServerConnectionFailed` is invoked when the client library is unable to establish a connection to the Diffusion server
- `DiffusionServerDisconnected` is invoked when an established connection to the Diffusion server is lost

Because these events are invoked synchronously, do not perform any long-running or blocking operations on these event threads. If you want to make another connection attempt after one of these events is invoked, create another thread to perform this task.

(Deprecated) Event publisher API

The ExternalClient API provides the ability to connect to a Diffusion server as an external client from within any .NET application.

> **Caution:**
>
> The event publisher API is deprecated. Interactions using this API are still supported in Diffusion version 5.1, but this feature is replaced in later versions.

You can use the Unified API to develop a control client that has equivalent capabilities to an event publisher.

There is a single class called `EventPublisher` which can be instantiated with the required connection details and used to make an actual connection. Messages can be sent to the server and each is routed to the publisher that owns the message topic.

The connection can be made only over DPT and as such `ServerDetails` cannot be specified.

The port specified to the connection must correspond to an event publisher connector for a Diffusion server running on the specified host.

For a detailed description of the API see the issued documentation (in docs directory).

The following example shows the simple case of establishing a connection and sending a single message, though in reality an event publisher sends many messages over a period of time.

```
public class EventPublisherApplication : IEventPublisherListener
   {
        #region Fields
        privatereadonlyEventPublisherthePublisher;
        #endregion// Fields
        #region Constructor
        publicEventPublisherApplication()
        {
            // Instantiate the event publisher
            thePublisher =
                newEventPublisher( "localhost", "3098", this );
            thePublisher.Connect();
        }
        #endregion// Constructor
        ///<summary>
        /// Notification of connection.
        ///
        /// This is called when a connection to an event publisher is
 established.
        ///</summary>
        ///<param name="connector">The server connector.</param>

 publicvoidServerConnected(DiffusionEventPublisherConnectorconnector)
        {
            ITopicMessagemessage =
                connector.CreateDeltaMessage( "MyTopic" );
            connector.Send( message );
        }
        publicvoidServerDisconnected(
            DiffusionEventPublisherConnectorconnector,
            ServerClosedEventArgsargs )
        {
            Debugging.StackTraceDebugPrint(
                "Event publisher connection closed: " + connector );
        }
        publicvoidMessageFromServer(
            DiffusionEventPublisherConnectorconnector,
            MessageImplmessage )
        {
            Debugging.StackTraceDebugPrint(
                "Message from server: " + message );
        }
    }
```

JavaScript API

The JavaScript API provides web developers with a simple means of connecting to and interacting with a Diffusion server from within a web browser. The API takes care to select the most appropriate underlying transport from those available.

For the list of supported web browsers, see Browsers supported on page 118.

Using the JavaScript API

The JavaScript client library is located in the `clients/js` directory of the Diffusion installation. Two versions are provided, an uncompressed file, and a minimized file with the extension `min.js`.

To enable the Diffusion client, the web page must load `diffusion-js.js`:

```
<script type="text/javascript" src="clients/js/diffusion-js.js"></script>
```

Dependent transport files

The JavaScript client depends on other files located in the same directory: `clients/js/diffusion-client.swf` and `clients/js/diffusion-client.xap`. These files provide Flash and Silverlight transport capabilities, respectively. Removing these files prevents the JavaScript client from using these transports.

You can configure the JavaScript client to point to specific versions of both the Flash and the Silverlight transports. This is available through the connection details.

```
var connectionDetails = new DiffusionClientConnectionDetails();

connectionDetails.libPath = "/lib/js/diffusion";
connectionDetails.libFlashPath = "diffusion-flash.swf";
connectionDetails.libSilverlightPath = "diffusion-silverlight.xap";
```

Connection details

The `DiffusionClientConnectionDetails` object has over 20 attributes that change the way that the client behaves. Any attributes that are not set are provided with default values when used.

You can provide an anonymous object instead of instantiating a new `DiffusionClientConnectionDetails` object.

Connecting

A Diffusion client is a singleton with global scope. It can be called from anywhere. To connect to Diffusion, call the connect method. The method takes two parameters, first is the connection details and the optional second object is the client credentials. The following example is using an anonymous connection object:

```
 DiffusionClient.connect({
debug : true,
onCallbackFunction : function(isConnected) {
 console.log("Diffusion client is connected: " + isConnected);
 }
})
```

Credentials

Credentials can either be supplied on the connect method or set separately using the `DiffusionClient.setCredentials(...)` method. These credentials are used for all transports that are used to connect to Diffusion. The `DiffusionClientCredentials` object is a simple one of username and password attributes.

```
// Connect with supplied credentials
     DiffusionClient.connect({...}, {
      username : "foo",
      password : "bar"
     });
```

```
// Connect, and send credentials later
     DiffusionClient.connect({
      onCallbackFunction : function() {
       DiffusionClient.sendCredentials({
        username : "foo",
        password : "bar"
       });
      }
     });
```

Events

The connection details have attributes that are listeners for certain events. If these are set in the connection object, they are called when these events happen.

Table 79: JavaScript functions called on events

Function	Description
`onDataFunction`	This function is responsible for handling messages from Diffusion. This function is called with an argument of `WebClientMessage`. This function is called even if there is a topic listener in place for a topic.
`onBeforeUnloadFunction`	This function is called when the user closes the browser or navigates away from the page.
`onCallbackFunction`	This function is called when Diffusion has connected, or exhausted all transports and cannot connect. This function is called with a boolean argument.
`onInvalidClientFunction`	This function is called when an invalid Diffusion operation is called, for instance if `Diffusion.subscribe` is called before `Diffusion.connect`
`onCascadeFunction`	This function is called when the client cascades transports. The function is called with an argument of the {String} transport name or NONE if all transport are exhausted.
`onPingFunction`	This function is called with an argument of `PingMessage` when the ping response has been returned from the server.
`onAbortFunction`	This function is called when the Diffusion server terminates the client connection (or the connection has been banned).

Function	Description
`onLostConnectionFunction`	This function is called when the client loses connection with the Diffusion server
`onConnectionRejectFunction`	This function is called when the client connection is rejected by the Diffusion server because of incorrect credentials.
`onMessageNotAcknowledgedFunction`	This function is called when a message that is requested as Acknowledge did not respond in time.
`onServerRejectedCredentialsFunction`	This function is called after a `DiffusionClient.sendCredentials` and the server rejected the credentials.
`onTopicStatusFunction`	This function is called if the status of a subscribed topic changes.

Receiving messages

The `onDataFunction` with a class called `WebClientMessage` contains only one message even if the messages sent from the Diffusion server are batched. If this is the case, this method is repeatedly called until all of the messages are exhausted. The `WebClientMessage` class wraps the message sent from the Diffusion server with utility methods like `isInitialTopicLoad()` and `getTopic`. See the jsdoc for the full list of utility methods.

Sending messages

There are two ways of sending messages to the Diffusion server:

- The `DiffusionClient.send(topic, message)` method
- The `sendTopicMessage` method

If user headers are required, it is best to use the `TopicMessage` class. THe following example shows how to send a message using the `TopicMessage` class.

```
var topicMessage = new TopicMessage("Echo", "This is a message");
topicMessage.addUserHeader("Header1");
topicMessage.addUserHeader("Header2");

DiffusionClient.sendTopicMessage(topicMessage);
```

Subscribing and unsubscribing

To subscribe and unsubscribe use the `DiffusionClient.subscribe(topic)` and `DiffusionClient.unsubscribe(topic)` methods respectively. The parameter can be a topic, a topic selector, or a comma-delimited list of topics

Topic listeners

During the lifetime of the connection, it might be required to have modular components that are notified about topic messages. These are topic listeners. A topic listener calls a supplied function with a `WebClientMessage` object when the topic of the message matches the pattern supplied. It is also worth noting that the `onDataMessage` function is called as well as the topic listener function. You can have many topic listeners on the same topic pattern if required. For example, if you want to be notified about a particular topic, you can use the following example code:

```
DiffusionClient.addTopicListener(""^Logs$", onDataTradeEvent);
```

Note:

The characters `^ $` are regular expression characters. For more information, see Regular expression. The preceding code example means that the listener is only interest in receiving the message event if the topic is Logs. If the following example code is used, any topic name that has Logs in it matches.

```
var listenerRef = DiffusionClient.addTopicListener("Logs",
 onLogEvent, this);
```

Retain the listener reference if you want to remove the topic listener at a later date.

Failover

The JavaScript client does not support autofailover. You can still implement this using the `onLostConnectionFunction`.

Special Features
Paged topic data handling

Where paged topic data is in use at the server there are features within the client API which simplify the handling of messages to and from such a topic.

Reconnecting with the JavaScript API

The JavaScript API supports reconnection. If you have reconnection enabled and you lose your connection to a server, you can reestablish it, using the same client ID and with the client subscriptions preserved.

The JavaScript API listens for pings from the server and raises and event if the connection to the server is lost.

To enable the liveness monitor, set the `enableLivenessMonitor` parameter to true inside the client connections details. For example:

```
var connectionDetails = { ...
    enableLivenessMonitor : true,
... };

DiffusionClient.connect(connectionDetails);
```

The Diffusion server sends out pings at regular intervals. The length of this interval is configured at the server by using the `system-ping-frequency` element in the `Connectors.xml` configuration file.

The liveness monitor in the ActionScript client library listens for the pings from the server and uses them to estimate the ping interval. The liveness monitor takes an average of the time between the pings it receives to estimate the ping interval. It revises this estimation each time it receives a ping, until it has received ten pings. After ten pings the liveness monitor has obtained the estimated ping interval that it uses for the rest of the client session.

If the liveness monitor does not receive a ping within a time interval equal to twice the length of the estimated ping interval, it considers the connection lost and raises a connection lost event.

Connection lost events can be raised by the liveness monitor or triggered by other events, such as an unexpectedly closed connection.

You can implement an event listener in your client that listens for a connection lost event and reacts to it by using the `reconnect()` method to reestablish the connection.

> **Reconnection example**
>
> To reconnect after you lose connection, you must use the `reconnect()` method. You cannot reconnect an aborted client.

The following code shows how to setup an event listener for connection events, if the connection has been lost how to reconnect and how to tell if you have successfully reconnected the client.

```
var connectionDetails = { onCallbackFunction :
 function( isConnected, isReconnect)
    { if( !isConnected && isReconnect )
 DiffusionClient.reconnect(); }
    }, onLostConnectionFunction : function()
    { DiffusionClient.reconnect(); }
};

DiffusionClient.connect(connectionDetails);
```

Related Links

Client reconnection on page 504
You can configure the client reconnection feature by configuring the connectors at the Diffusion server to keep alive the client session.

http://docs.pushtechnology.com/docs/5.1.0/js/index.html

Service topic data in JavaScript

The JavaScript API provides a basic interface for using service topics.

The API consists of a service topic handler to process responses and using the generic `DiffusionClient.command(...)` method to send service requests.

The common sequence to follow is:

1. Add a topic listener, to capture the service topic load message
2. Subscribe to the service topic
3. With the ITL from the service topic create a service topic handler
4. Remove the topic listener
5. Send command messages to the service
6. Process any response in the function passed to the handler

To create a handler using the `DiffusionClient.createServiceTopicHandler(TopicMessage, function)` you must pass in the ITL of the service topic and the function that is called when a service response is received. This function will be called with a `CommandMessage` as an argument.

To make service requests you must use the `DiffusionClient.command(string,string,TopicMessage)` method to send command messages. The first string is the command to send. The second string is a correlation ID for the response. The TopicMessage is the message sent to the client with the correct topic and any additional headers or payload you want to send in the request.

The messages you send and process must conform to the Service topic Protocol.

Use an ordinary topic listener to get the ITL to create the service topic handler. This listener is not required for any subsequent message processing and you are encouraged to remove it after you have the ITL.

You must generate a unique value for the correlation ID.

Paged topic data in JavaScript

The JavaScript API provides an interface for using paged topics.

The API contains the following classes:

`PagedTopicHandler`
: Provides methods that enable you to change and navigate the page view.

`PagedTopicListener`
: Provides callbacks for when an action is performed on a page or the topic.

`PageStatus`
: Contains values that describe the status of a page or topic.

For more information, see JavaScript Classic API documentation.

Handler methods

The following example code shows some of the handler methods wrapped in functions you can use to add buttons to a client's user interface.

```
// Get the next page in the topic
function next() {
 handler.next();
}

// Get the previous page in the topic
function prior() {
 handler.prior();
}

// Get the first page in the topic
function first() {
 handler.first();
}

// Get the last page in the topic
function last() {
 handler.last();
}

// Close the paged view of the topic
function pagedclose() {
 handler.close();
}
```

Listener methods

The following example code creates a `PagedTopicHandler` and implements the listener methods `add`, `page`, `ready`, `statusChanged`, and `update`.

```
var connectionDetails = {
 onDataFunction : function() {
 },
 onCallbackFunction : function() {
  // Creates the handler for a topic
  DiffusionClient.createPagedTopicHandler(topic, {
   ready : function(handler) {
    // Add here the code you want to run when the handler
    // is created.

    // For example, open a view that is 20 lines long and
    // contains the first page of data:
    handler.open(20, 1);
   },
   page : function(handler, status, lines) {
    // Add here the code you want to run when the page is
    // loaded.

   },
   update : function(handler, status, index, line) {
```

```
    // Add here the code you want to run when a line on
    // the current page is updated.

    // For example, refresh the page.
    handler.refresh();
   },
   statusChanged : function(handler, status) {
    // Add here the code you want to run when the status
    // of the current page changes.

    // For example, check whether the page is dirty and
    // refresh the page if this is true.
    if (status.isDirty){
     handler.refresh();
    }
   },
   add : function(handler, status, lines) {
    // Add here the code you want to run when a line is
    // added to the current page.
   }
  });
 },
 debug : true
};
```

Related Links

Paged topic data on page 167
Paged topic data provides the functionality of a paged topic. The data is formatted in lines. A client can view the data as pages made up of one or more lines and can page forward and backward through the data.

ActionScript API

The ActionScript API is bundled in a library called `clients/flex/diffusion-flex.swc`.

This can be embedded into a Flex/Flash or Air application. Full asdoc is issued with the product so the sections below provide a brief outline of the uses for the classes and examples of their use. The ActionScript library is based on the event model. There are many different types of events that a DiffusionClient dispatches. These must be registered before notification happens.

Diffusion also provides a debug-friendly version of the library: `diffusion-flex-debug-x.x.x.swc`, where *x.x.x* is the Diffusion version number, for example 5.1.0. This version is larger, but you can embed it into you application to receive additional information from any stack traces.

Using the ActionScript API

DiffusionClient (`com.pushtechnology.diffusion.DiffusionClient`) is the main class that is used. This class enables the user to set all of the connection and topic information.

DiffusionClient

Connection example

```
// Get a new DiffusionClient
theClient = new DiffusionClient();

// Set up the transport mode
theClient.setTransportMode(DiffusionClient.CASCADE);

// Set everything to enable the cascading
theClient.setTopic("Trade");
```

```
theClient.setHost("192.168.52.2");
theClient.setPort(3095);
theClient.setURL("http://localhost:8080");

// Add the listeners
theClient.addEventListener(DiffusionConnectionEvent.CONNECTION,
onConnection);
theClient.addEventListener(DiffusionMessageEvent.MESSAGE,
onMessages);
theClient.addEventListener(DiffusionTraceEvent.TRACE,
onTrace);
theClient.addEventListener(DiffusionExceptionEvent.EXCEPTION,
onException);
theClient.addEventListener(DiffusionPingEvent.PING, onPing);

// Connect
theClient.connect();
```

Setting credentials

If credentials are required by the Diffusion server then use the `setCredentials` method on the `DiffusionClient` class. The `DiffusionClientCredentials` class takes a constructor argument of username and password. Please bear in mind, that these are only tokens and can contain any information that the `AuthorisationHandler` requires.

```
var credentials:DiffusionClientCredentials = new
DiffusionClientCredentials(username, password);
theClient.setCredentials(credentials);
```

Connection event

The connection event contains information about the success of the connection attempt. Below is a coding example of the possibilities for the connect event.

```
public function onConnection(event:DiffusionConnectionEvent) : void {
 if(event.wasConnectionRejected()) {
  theClientIDBox.text = "Connection Rejected by Diffusion Server";
 }else if(event.wasClientAborted()) {
  theClientIDBox.text = "Connection aborted";
 } else if(event.isConnected()) {
  theClientIDBox.text = event.getClientID();
  theConnectedTransportLabel.text = theClient.getTransportMode();
 } else {
  theClientIDBox.text = "Connection failed " + event.getErrorMessage();
 }
}
```

You can receive a connection event after you have successfully connected, which might be because of a lost connection, or in the case of client aborted the Diffusion server has deliberately closed the client connection. This normally means that a publisher has aborted the connection and the client must not try and connect again.

onMessage event

When messages arrive from the Diffusion server on a subscribed topic, the `DiffusionMessageEvent` is dispatched. Contained in the event is a TopicMessage object `TopicMessage` (com.pushtechnology.diffusion.TopicMessage). This class contains helper methods that surround the message itself, like getTopic() and isInitialTopicLoad. For more information, see the API documentation.

```
public function onMessages(event:DiffusionMessageEvent) : void {
 var message:TopicMessage = event.getTopicMessage();
```

```
...
```

Subscriptions

Once the client has connected, you can issue subscribe and unsubscribe commands. The subscribe and unsubscribe methods take a string format, that can be a topic selection pattern, a list of topics that are comma delimited or a single topic.

Send

Once connected a client can send messages to the Diffusion server on a particular topic. To do this, use the send method.

```
theClient.send("Fred","Hello publisher that looks after Fred");
```

In the example above, the publisher that looks after topic Fred receives a `messageFromClient` notification. If a message with user headers or encoding is required, you must use the `sendTopicMessage` method. A TopicMessage (`com.pushtechnology.diffusion.TopicMessage`) allows for the setting of user headers and message encoding

Ping

The client can ping the Diffusion server. To receive the Ping response, the listener is added to the client.

```
theClient.addEventListener(DiffusionPingEvent.PING, onPing);
```

The resulting ping event has two attributes in it, firstly the time taken to do a round trip from the client to the Diffusion server and back again. The second attribute is how many items are currently in the client queue at the server. This information enables the client to get some vital connection information. It is down to the implementation of the client to specify the ping frequency, if at all required.

Topic listeners

During the life time of the connection, it might be required to have modular components notified about topic messages – these are topic listeners. A topic listener calls a supplied function with a TopicMessage object when the topic of the message matches the pattern supplied. The topic listeners are called in the order that they are added, and before the default DiffusionMessageEvent.MESSAGE, that is called as well as the topic listener event. You can have many topic listeners on the same topic pattern if required. The function supplied in charge of processing the message can signal that a message is consumed, returning TRUE. In this case, this message is not relayed to subsequent TopicListeners and the default listener. For example, if you want to be notified about a particular topic, use the following code:

```
var listenerRef:String theClient.addTopicListener("^Logs$",
 theLogsDataGrid.onMessage);
```

Note the syntax here, the ^ $ are regex pattern strings, the above means that the listener is only interested in receive the message event if the topic is Logs. If the following was issued.

```
var listenerRef:String theClient.addTopicListener("Logs",
 theLogsDataGrid.onMessage);
```

Any topic name that has “Logs” in it matches. You must store the reference to remove the topic listener at a later date.

Timed topic listeners

A timed topic listener calls a supplied function with an array of topicMessage objects when the topic of the message matches the pattern, and only if the time supplied by the arguments has expired. Otherwise, the TopicMessage is stored until the time expired.

> **Note:**
>
> The function in charge of processing the message cannot determine if a message is consumed as you can do in a topic listener. For example, if you want to be notified about a particular topic, use the following code:
>
> ```
> var timedListenerRef:String theClient.addTimedTopicListener("^Logs
> $", theLogsDataGrid.onMessage, 2000, false);
> ```

The third parameter is the frequency at which the function supplied is called. The optional fourth parameter can be set if this function must be called, even if no messages are stored.

Failover

The ActionScript client supports autofailover. For more information, see the ActionScript failover documentation.

Special features

Paged topic data handling	Where paged topic data is in use at the server there are features within the client API which simplify the handling of messages to and from such a topic.

Reconnecting with the ActionScript API

The ActionScript API supports reconnection. If you have reconnection enabled and you lose your connection to a server, you can reestablish it, using the same client ID and with the client subscriptions preserved.

Liveness monitor

The ActionScript API implements a liveness monitor that listens for pings from the server and raises an event if the connection to the server is lost.

Before you make a connection to the Diffusion server, enable the liveness monitor by using the `enableLivenessMonitor()` method. For example:

```
client.enableLivenessMonitor(true);
  client.connect();
```

The Diffusion server sends out pings at regular intervals. The length of this interval is configured at the server by using the `system-ping-frequency` element in the `Connectors.xml` configuration file.

The liveness monitor in the ActionScript client library listens for the pings from the server and uses them to estimate the ping interval. The liveness monitor takes an average of the time between the pings it receives to estimate the ping interval. It revises this estimation each time it receives a ping, until it has received ten pings. After ten pings the liveness monitor has obtained the estimated ping interval that it uses for the rest of the client session.

If the liveness monitor does not receive a ping within a time interval equal to twice the length of the estimated ping interval, it considers the connection lost and raises a `DiffusionConnectionEvent` whose `hasLostConnection()` method returns true.

You can implement an event listener in your client that listens for this event and reacts to it by using the `reconnect()` method to reestablish the connection.

Warning:

The liveness monitor relies on server pings being received at regular intervals. If the server pings the client in addition to the regular pings, these additional pings can cause the liveness monitor to make an incorrect estimate of the ping interval. Because this incorrect estimate is shorter than the correct ping interval, this can cause the liveness monitor to incorrectly consider a connection lost.

To avoid this problem, if you are using the liveness monitor, ensure that you do not ping the client from a publisher or from the Introspector.

Reconnection example

To reconnect, you must use the `reconnect` method when you lose a connection. You cannot reconnect an aborted client.

The following code shows how to setup an event listener for connection events, if the connection has been lost how to reconnect and how to tell if you have successfully reconnected the client.

```
var client:DiffusionClient = createClient();

function onConnectionEvent(event:DiffusionConnectionEvent) {
 if (event.hasLostConnection()) {
  client.reconnect();
 }
 else if (event.isConnected()) {
  if (event.isReconnected()) {
   // Successful reconnection
  }
 }
}

function createClient():DiffusionClient {
 var client:DiffusionClient = new DiffusionClient();
 client.addEventListener(DiffusionConnectionEvent.CONNECTION,
 onConnectionEvent);
 return client;
}
```

Related Links

Client reconnection on page 504
You can configure the client reconnection feature by configuring the connectors at the Diffusion server to keep alive the client session.

http://docs.pushtechnology.com/docs/5.1.0/flex/index.html

Silverlight API

The Silverlight API is bundled in an assembly called `clients/silverlight/PushTechnology.Transports.dll`. This can be embedded into a Silverlight application.

Full API documentation is issued with the product, so the sections below provide a brief outline of the uses for the classes and examples of their use. The Silverlight library is based on an asynchronous event model. There are a few events that the client object invokes. The client must subscribe to these events before notification can happen.

Using the Silverlight API

The `DiffusionClient` class is the main class that is used. This class enables the user to set all of the connection and topic information.

Instantiation and connection example

```
 theClient = new DiffusionClient( Dispatcher );

   // Instantiate the server details object and the initial
 topic to subscribe to
   ServerDetails details = new ServerDetails( "http://
localhost:8080", "SpotOnly" );

   // Add the server details to the client
   theClient.AddServerDetails( details );

            // Add the event listeners
            theClient.ConnectionStatus +=
 DiffusionConnectionStatus;
            theClient.MessageReceived +=
 theClient_MessageReceived;

            // Now connect
            theClient.Connect();
```

Setting credentials

If credentials are required by the Diffusion server then use the `Credentials` property on the `DiffusionClient` class. The `DiffusionClientCredentials` class takes a constructor argument of `userName` and `password`. Please bear in mind that these are only tokens and can contain any information that the `AuthorisationHandler` requires.

```
theClient.Credentials = new DiffusionClientCredentials(
    "username", "password" );
```

Rejection of credentials event

If the credentials are rejected by the Diffusion server, a `ServerRejectedCredentials` event is fired. This can be subscribed to by using the following code:

```
 theClient.ServerRejectedCredentials +=
    ServerRejectedCredentials;
```

Message not acknowledged event

When a message is created with the "acknowledge" flag, this event is fired when a message is not acknowledged by the Diffusion server within the specified time period. This can be subscribed to to by using the following code:

```
theClient.MessageNotAcknowledged += MessageNotAcknowledged;
```

The ConnectionStatus event

The `ConnectionStatus` event contains information about whether the connection was successful. Here follows an example of the usage of this event.

```
  /// <summary>
        /// Called when the connection state to Diffusion changes.
        /// </summary>
        /// <param name="sender"></param>
```

```
        /// <param name="e"></param>
        void DiffusionConnectionStatus(
            object sender,
            DiffusionConnectionStatusEventArgs e )
        {
            switch( e.StatusType )
            {
                case DiffusionConnectionStatusType.ConnectionFailed:
                {
                    Dispatcher.BeginInvoke( () =>
                        MessageBox.Show( "Unable to connect to Diffusion.
Diffusion reports: " + e.ExtraData,
                            "Connection failed", MessageBoxButton.OK ) );
                }
                break;

                case DiffusionConnectionStatusType.ConnectionReset:
                {
                    Dispatcher.BeginInvoke( () =>
                        MessageBox.Show( "The connection to Diffusion has
been reset.  Diffusion reports: " + e.ExtraData,
                            "Connection failed", MessageBoxButton.OK ) );
                }
                break;
            }
        }
```

Note: You can receive a `ConnectionStatus` event after you have successfully connected; it might be because of a lost connection, or in the case of `ConnectionAborted`, the Diffusion server has closed the client connection. This normally means that a publisher has aborted the connection and the client must not try to connect again.

The MessageReceived event

When messages arrive from the Diffusion server on a subscribed topic, the `MessageReceived` event is fired. This event contains a sender and a `TopicMessageEventArgs` object which itself contains a `TopicMessage` object which can be interrogated to discover the contents of the received message.

The TopicStatusMessageReceived event

When the status of a topic changes on the Diffusion server, the `TopicStatusMessageReceived` event is fired. This event contains a sender and a `TopicStatusMessageEventArgs` object which contains the alias of the topic on which the status has changed. Currently, only the notification of the removal of a topic is implemented.

Subscriptions

After the client has connected, you can issue `Subscribe` and `Unsubscribe` commands. These commands take string arguments which can be a topic selection pattern, a list of topics that are comma-delimited, or a single topic.

Sending non-encoded topic messages

Once the client has connected, it can send messages to the Diffusion server on a particular topic. To do this, use either the `Send` or `SendTopicMessage` methods on the DiffusionClient object, as shown in the following code:

```
  theClient.Send( "Fred", "Hello, publisher that looks after Fred" );

            TopicMessage message =
                new TopicMessage( "Fred", "Hello, publisher that looks
after Fred" );
```

```
                theClient.SendTopicMessage( message );
```

> **Note:** The `TopicMessage` itself contains methods to set (for instance) user headers and encoding, or the convenience methods described below can handle the alternate encoding scenarios.

In the above examples, the publisher that looks after the topic `Fred` receives a `messageFromClient` notification internally.

Sending an encrypted topic message

Sending an encrypted topic message is achieved by calling the `SendTopicMessageEncrypted` method on the `DiffusionClient` object by using the following code:

```
  theClient.SendTopicMessageEncrypted( new TopicMessage( "Fred", "Hello,
publisher that looks after Fred" ) );
```

This sets the relevant encoding flags on the message itself, and the message will be encrypted immediately prior to sending to the Diffusion server.

> **Note:** Because of the limitations of HTTP-based transports, attempting to send a message of this type results in a non-encoded message being sent.

Sending a compressed topic message

Sending a compressed topic message is achieved by calling the `SendTopicMessageCompressed` method on the `DiffusionClient` object by using the following code:

```
  theClient.SendTopicMessageCompressed( new TopicMessage( "Fred", "Hello,
publisher that looks after Fred" ) );
```

This sets the relevant encoding flags on the message itself, and the message will be compressed immediately prior to sending to the Diffusion server.

> **Note:** Because of the limitations of HTTP-based transports, attempting to send a message of this type results in a non-encoded message being sent.

Sending a Base64-encoded topic message

Sending a Base64-encoded topic message is achieved by calling the `SendTopicMessageBase64` method on the `DiffusionClient` object to using the following code:

```
theClient.SendTopicMessageBase64( new TopicMessage( "Fred", "Hello,
 publisher that looks after Fred" ) );
```

This sets the relevant encoding flags on the message itself, and the message will be Base64-encoded immediately prior to sending to the Diffusion server.

Ping

A client can ping the Diffusion server. To process the ping response, the user monitors the `MessageReceived` event and checks for a message type of `PingServer`, as shown in the following code:

```
    private void HandleServerPingMessage( TopicMessageEventArgs e )
  {
   var message = e.Message as PingMessage;

   if( message != null )
   {
    tbElapsedTime.Text = message.ElapsedTime.ToString();
    tbQueueSize.Text = message.QueueSize.ToString();
   }
```

```
}
```

Fetch

Using the `fetch` method, a client can send a request to the Diffusion server for the current state of a topic, which returns a state message to the client. A client can do this even if not subscribed to the topic.

Topic listeners

During the lifetime of the connection, it might be required to have modular components that get notified about topic messages – these are known as topic listeners. A topic listener calls a supplied function with a `TopicMessage` object when the topic of the message matches the pattern supplied. It is also worth noting that the `OnMessageReceived` event is called as well as the topic listener event itself.

You can have many topic listeners on the same topic pattern if required. For example, if you want to be notified about a particular topic, use the following code:

```
instrumentListener = theClient.AddTopicListener(
            "^SpotOnly$", ProcessInstruments, this );
```

Note: The "^" and "$" characters are regular expression pattern strings; the above means that the listener is only interested in receiving the message if the topic is "SpotOnly".

Enabling JavaScript method invoking

To call JavaScript functions (and they are permitted to do so by the Silverlight runtime), use the following method call:

```
theClient.InitialiseJavaScriptMethodInvoking( HtmlPage.Window );
```

Listening to internal transport debug messages

To subscribe to the internal log tracings of the Silverlight API, the user can subscribe to the `DiffusionTraceEvent` on the `DiffusionClient` object, as shown in the following code:

```
theClient.DiffusionTraceEvent += theClient_DiffusionTraceEvent;
```

This enables the user to monitor all internal debug messages within the Silverlight API.

iOS API

The file `clients/ios/diffusion-ios.zip` contains the static libraries and header files that comprise the Diffusion iOS API.

Full API documentation is issued with the product so the sections below provide a brief outline of the purpose of the classes and examples of their use. The iOS Library uses the delegate model. There are many different types of events that a DFClient object dispatches. These must be registered before notification takes place

The API documentation is included also as an XCode docset. Once installed into XCode the Diffusion client for iOS can be browsed within the XCode Organizer/Documentation browser.

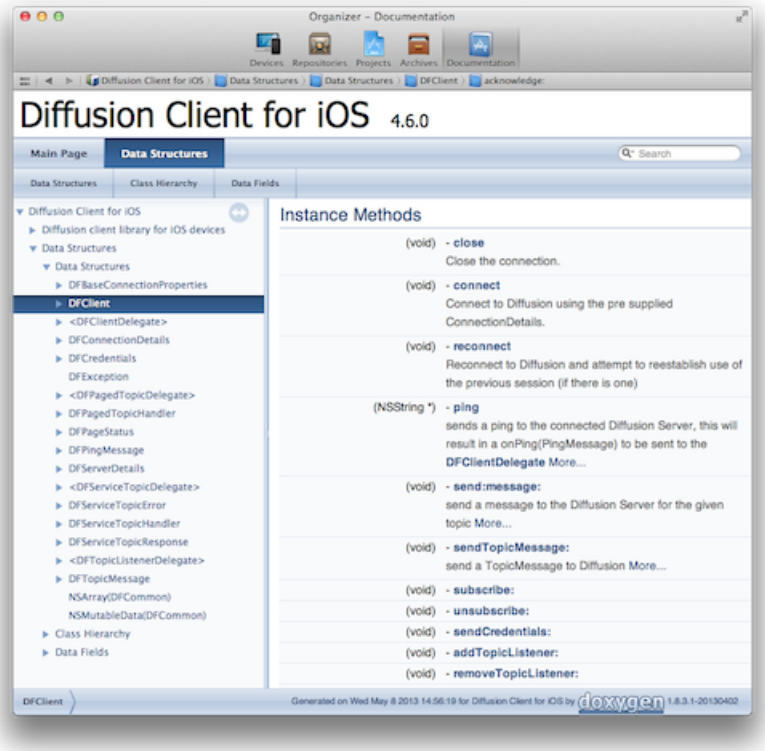


Figure 24: XCode documentation browser

DFClient

The `DFClient` class is the main class that is used. This class enables the user to set all of the connection and topic information.

Connection example

```
NSURL *serverURL = [NSURL URLWithString:@"dpt://
demo.pushtechnology.com:80"];
DFServerDetails *server = [[[DFServerDetails alloc]
 initWithURL:serverURL ] autorelease];
DFConnectionDetails *cnxDetails = [[[DFConnectionDetails alloc]
 initWithServer:server topics:@"Echo" andCredentials:nil] autorelease];

dfClient = [[DFClient alloc] init];
dfClient.connectionDetails = cnxDetails;
dfClient.delegate = self; // Presumes that this class implements the
 DFClientDelegate protocol
[dfClient connect];
```

Diffusion Delegate

The Diffusion Delegate class is a custom class that must adhere to the DFClientDelegate protocol. The protocol consists of the following methods

```
/**
 Protocol implemented by classes wishing to receive notification from
 Diffusion.
 Notification primarily of new messages and the state of the connection to
 the server.
 */
@protocol DFClientDelegate
```

```
/**
 * This method is called when the DFClient tries to connect, if the
 connection is made, isConnected is true
 * @param isConnected
 */
- (void) onConnection:(BOOL) isConnected;

/**
 * This method is called when the DFClient has lost connection to the
 Diffusion server
 */
- (void) onLostConnection;

/**
 * This method is called when the Diffusion server has terminated the
 connection (barred)
 */
- (void) onAbort;

/**
 * This method is called when a message has been received from the
 Diffusion server.
 * This method is called as well as any topicListeners that might match
 the topic.
 */
- (void) onMessage:(DFTopicMessage *) message;

/**
 * This method is called on receipt of the ping request
 * @see DFClient
 * @param message PingMessage
 */
- (void) onPing:(DFPingMessage *) message;

/**
 * This method is called after a send credentials message, and the server
 rejected the credentials
 * @see DFClient
 */
- (void) onServerRejectedConnection;

/**
 * This method is called if the server did not respond to an Ack message
 in time
 * @see TopicMessage
 */
- (void) onMessageNotAcknowledged:(DFTopicMessage *) message;

/**
 The list of DFServerDetails object has been exhausted, and no connection
 can be placed.
 Once this method is called the set of DFServerDetails is reset and
 further connections can be placed. In most simple scenarios where
 there is only one DFServerDetails object in the DFConnectionDetails
 object call method [client connect] here.
 @param client DFClient that has exhausted its set of DFServerDetails
 object from the DFClientDetails object.
 */
-(void)onConnectionSequenceExhausted:(DFClient*)client;

@optional

/**
 Conveys news from the Diffusion server that the named topic no longer
 exists
 */
-(void)onTopicRemoved:(NSString*) topicName;
```

```
/**
 The given DFServerDetails object has been selected for connection.
 @param details Details object that has been chosen.
 @param client DFClient that has chosen this DFServerDetails
 */
-(void)onConnectionDetailsAcquired:(DFServerDetails*)details forClient:
(DFClient*)client;
```

You can receive an `onConnection` event after you have successfully connected, is might be due to a lost connection.

Credentials

When credentials are required, use the `credentials` property on the DFClient class. It is required that the user create a DFCredentials class and then set it on the client before calling connect.

onMessage event

When messages arrive from the Diffusion server on a subscribed topic, the `onMessage` method is called on the delegate provided. The message is wrapped in a class called `TopicMessage`. This class contains helper methods that surround the message itself, such as the topic and isInitialLoad properties. For more information, see the API documentation.

Subscriptions

Once the client has connected, you can issue subscribe and unsubscribe commands. The subscribe and unsubscribe methods take a string format, that can be a topic selection pattern, a list of topics that are comma delimited or a single topic.

Send

Once connected, you can send messages to the Diffusion server on a particular topic. To do this, use the send method.

```
[mClient send:"Fred" :"Hello publisher that looks after Fred"];
```

In the example above, the publisher that looks after topic Fred receive a `messageFromClient` notification. If sending a message with user headers, use the `sendTopicMessage` method. A TopicMessage allows for the setting of user headers

Ping

You can ping the Diffusion server. The delegate is notified by the onPing method. The resulting ping event has two attributes in it, firstly the time stamp of the request. The second attribute is how many items are currently in the client queue at the server. This information enables the client to get some vital connection information. It is down to the implementation of the client to specify the ping frequency, if at all required.

Topic listeners

During the lifetime of the connection, it might be required to have modular components notified about topic messages – these are topic listeners. A topic listener calls a supplied method with a TopicMessage class when the topic of the message matches the topic name. Please note for performance the iOS topic listeners do not have regular expression patterns but topic name matching. You can have many topic listeners on the same topic pattern if required. For example, if you want to be notified about a particular topic, issue the following listener:

```
[mClient addTopicListener:aTopicListener];
```

Where aTopicListener implements the protocol `DFTopicListenerDelegate` which is shown below.

```
/**
 Protocol for receiving messages from a particular topic.
 */
@protocol DFTopicListenerDelegate

/**
 * This method is called if the TopicMessage matches the message received
 from Diffusion
 *
 * @param message
 * @return YES if the message is consumed and must not be relayed to
 subsequent DFTopicListenerDelegate, nor the default listener.
 */
- (BOOL) onMessage:(DFTopicMessage *) message;

/**
 * This is the topic used to see if the message from Diffusion matches
 (equals) this String
 */
- (NSString *) getTopic;
```

Topic listeners can be removed by calling the removeTopicListener on the DFClient class

Installing the docset

Installing the Diffusion client for iOS docset

Procedure

1. Download the Diffusion Documentation Pack from http://download.pushtechnology.com
2. Unpack the ZIP file and locate the `iOS` folder.
3. With the `iOS` folder locate the file `com.pushtechnology.diffusion.ios.client.docset` and copy into one of the directories used by XCode for storing docsets (for example, `~/Library/Developer/Shared/Documentation/DocSets`) Alternatively, use the `Makefile` in the same directory and run `make install` to copy the same file into the same location.
4. Restart XCode if it is running

Android API

The Android API bundled in a library called `diffusion-android-x.x.x.jar`, where *x.x.x* is the version number, for example 5.1.0.

Full API documentation is issued with the product so the section below provides a brief outline of the uses for the classes and examples of their use. The DiffusionClient uses a listener model which closely resembles the external client API. The DiffusionConnectionListener interface is responsible for all of the event notifications from DiffusionClient

Using the Android API

DiffusionClient class is the main class that is used. ConnectionDetails beside ServerDetails classes enables the user to set all of the connection and topic information.

DiffusionClient

The following code shows an example of connection:

```
    // Get a new DiffusionClient
    theClient = new DiffusionClient();
```

```
    //Set the connection details
    ServerDetails serverDetails = new ServerDetails(new URL("dpt://
localhost:8080"));
    ConnectionDetails connectionDetails = new
 ConnectionDetails(serverDetails);
    theClient.setConnectionDetails(connectionDetails);

    // Make this listen to the DiffusionClient events
    theClient.setConnectionListener(this);

    // Connect
    theClient.connect();
```

DiffusionConnectionListener

The DiffusionConnectionListener interface consists of the following methods (for further information refer to the API documentation)

```
    /**
     * connected, called upon connection
     */
    void connected();

    /**
     * errorConnecting, called if there is an error connecting
     *
     * @param e
     */
    void errorConnecting(Exception e);

    /**
     * disconnected, called when the connection list lost
     */
    void disconnected();

    /**
     * connectionAborted, called when DiffusionServer has rejected the
 connection
     */
    void connectionAborted();

    /**
     * onMessage, called when a message has been received from Diffusion
     *
     * @param message
     */
    void onMessage(Message message);

    /**
     * onPingMessage, called when a ping response is received
     *
     * @param message
     */
    void onPingMessage(PingMessage message);

    /**
     * onMessageNotAcknowledged, called when an ack message has not been
 acknowledged by Diffusion
     *
     * @param message
     */
    public void onMessageNotAcknowledged(TopicMessage message);

    /**
     * onConnectionSequenceExhausted, called when the complete list of
 ServerDetails have been exhausted.
```

```
     */
    public void onConnectionSequenceExhausted();

    /**
     * onConnectionDetailsAcquired, called each time a ServerDetails
object is selected for connection.
     *
     * @param serverDetails
     */
    public void onConnectionDetailsAcquired(ServerDetails serverDetails);

    /**
     * onServerRejectedCredentials, called when Diffusion reject the
credentials.
     */
    public void onServerRejectedCredentials();
```

Credentials

When credentials are required, there are three ways to set the credentials. The ServerDetails, the ConnectionDetails and the DiffusionClient all have a `setCredentials` method. It is required that the user create a DiffusionCredentials object and pass it to one of these methods before calling connect. Use only one of these ways. If more than one way is used, the selection of the credentials to use is undefined. The Android API only supports sending credentials on connection.

onMessage event

When messages arrive from the Diffusion server on a subscribed topic, the `onMessage` is called on the delegate provided. The message is wrapped in a interface called `Message`. This interface contains helper methods that surround the message itself, like getTopic() and isInitialTopicLoad. For more information, see the API documentation.

Subscriptions

Once the client has connected, you can issue subscribe and unsubscribe commands. The subscribe and unsubscribe methods take a string format, that can be a topic selection pattern, a list of topics that are comma delimited or a single topic.

Send

Once connected a client can send messages to the Diffusion server on a particular topic. To do this, use the send method.

```
theClient.send("Fred","Hello publisher that looks after Fred");
```

In the example above, the publisher that looks after topic Fred receives a `messageFromClient` notification. If the message requires a user header, you must use the `sendTopicMessage` method. A TopicMessage allows for the setting of user headers.

```
    TopicMessage message = new TopicMessage("Fred");
    message.addUserHeader("myHeaders");
    message.setMessage("Hello publisher that looks after Fred");
    theClient.sendTopicMessage(message);
```

Ping

The client can ping the Diffusion server. The delegate is notified of the response by the onPing function. The resulting ping event has two attributes in it, firstly the time stamp of the request. The second attribute is how many items are currently in the client queue at the server. This information enables the client to get some vital connection information. It is down to the implementation of the client to specify the ping frequency, if at all required.

Topic listeners

During the life time of the connection, it might be required to have modular components that get notified about topic messages, these are topic listeners. A topic listener is called using its onMessage method with a message object when the topic of the message matches the topic name.

> **Note:** For performance the Android topic listeners do not have regular expression patterns but topic name matching.

You can have many topic listeners on the same topic pattern if required. For example, if you wanted to be notified about a particular topic, use the following code:

```
theClient.addTopicListener(topicListener);
```

Where `topicListener` implements the interface `TopicListener` which is shown below.

```
    /**
     * getTopic
     *
     * @return the topic that this listener is interested in. This does
     * not take regular expressions this must be an exact match
     */
    String getTopic();

    /**
     * onMessage
     *
     * @param message message which topic matches the getTopic method
     */
    void onMessage(Message message);
```

Topic listeners are removed by calling the removeTopicListener on the DiffusionClient class

Threading concerns

The Android DiffusionClient creates and dedicates a thread to listening to traffic from the Diffusion server and reacting to messages from it. Consequently methods on the DiffusionConnectionListener and DiffusionTopicStatusListener are executed in the same thread. Android does not allow background threads to interact with GUI controls, only the main thread is allowed to do so.

To overcome this, any non-main thread can pass a `java.lang.Runnable` to the main thread for execution via `android.view.View.post(Runnable action)`. For example:

```
/*
 * Called when the Diffusion connection is established
 */
public void connected() {
    // Post this Runnable to the GUI thread to change the display
    String statusTextStr = String.format( "Connected to %s:%d\n", HOST,
 PORT );
    setStatus( statusTextStr );
    Log.i( TAG, statusTextStr );
  }

   /**
    * Set the content of the status view
    * @param statusStr
    */
   private void setStatus(final String statusStr)
    {
// Pass a Runnable to the GUI thread to execute using one of its widgets
     outputView.post( new Runnable() {
         public void run() {
```

```
            statusText.setText( statusStr );
        }
    } );
}
```

User permissions in `Manifest.xml`

To establish a connection with the Diffusion server, Android devices must add the user permission `INTERNET` within the `Manifest.xml`. This permission allows applications to open network sockets.

C API

The C API is provided as a source distribution in the file `diffusion-c-classic-x.x.x.zip`, where `x.x.x` is the version number, for example 5.1.0. This file is located in the `clients/c` directory of your Diffusion installation.

The source builds to either a dynamic or shared library on UNIX systems. You can link it into your own C/C++ applications, or used as the foundation for creating Diffusion clients for other languages which can be extended through binary APIs.

Using the C API

Build the library using the make command and ensure that it is on your LD_LIBRARY_PATH.

Building

To build the library, type `make` in the source directory. This builds shared and static versions of the Diffusion library, `libdiffusion.so` and `libdiffusion.a` respectively. Additionally, a number of sample applications are also built.

Installation

Copy the libraries to a location on the user's `LD_LIBRARY_PATH` (or equivalent). Copy the header files, `diffusion.h` and `llist.h` to your C compiler's include path.

For example, on Ubuntu® 10.10 the following is appropriate:

```
# mkdir /usr/local/include/diffusion
# cp diffusion.h llist.h /usr/local/include/diffusion
# mkdir /usr/local/lib/diffusion
# cp libdiffusion.* /usr/local/lib/diffusion
# echo "/usr/local/lib/diffusion" > /etc/ld.so.conf.d/libdiffusion.conf
# ldconfig
```

Example usage

The following example shows how to connect to a Diffusion instance (no credentials):

```
DIFFUSION_CONNECTION *c = diff_connect("localhost", 8080, NULL);
if(c == NULL) {
 fprintf(stderr, "Failed to connect to Diffusion\n");
 return(-1);
}
```

The following example shows how to connect to a Diffusion instance (with credentials)

```
SECURITY_CREDENTIALS creds;
creds.username = strdup("smith");
creds.password = strdup("secret");
DIFFUSION_CONNECTION *c = diff_connect("localhost", 8080, &creds);
if(c == NULL) {
```

```
 fprintf(stderr, "Failed to connect to Diffusion\n");
 return(-1);
}
```

The following example shows how to request a subscription to a topic:

```
DIFFUSION_CONNECTION *c = diff_connect(...);

if(diff_subscribe(c, "Assets") == -1) {
 fprintf(stderr, "Failed to subscribe to topic\n");
 return(-1);
}
```

The following example shows how to use the event loop and callbacks:

```
void on_initial_load(DIFFUSION_MESSAGE *msg) {...}
void on_delta(DIFFUSION_MESSAGE *msg) {...}

...

DIFFUSION_CONNECTION *c = diff_connect(...);
diff_subscribe(...);

DIFFUSION_CALLBACKS callbacks;
DIFF_CB_ZERO(callbacks);  // Reset callback structure
callbacks.on_initial_load = &on_initial_load;
callbacks.on_delta = &on_delta;

diff_loop(c, &callbacks);
```

diffusion-wrapper.js

The Diffusion wrapper is a script which addresses a weakness in the Flash and Silverlight VMs – when running inside a web browser there is no provision for the execution of callback code when the user closes either the containing tab or the entire browser window.

Consequently there is no opportunity for the Diffusion client to inform the server that the client is willingly closing the connection, instead the connection is severed. This can result in various server warnings (dependent on the transport mechanism, that is DPT or HTTP) as well as maintaining the server-side reference to the client if any keep-alive properties are set.

The JavaScript environment in the hosting browser provides browser closing callbacks, and these are employed by DiffusionWrapper.js. By supplying the `setClientDetails` function with client and server info after a connection has been established within the Flash or Silverlight API, DiffusionWrapper attaches a JavaScript function to the `onbeforeunload` event that is triggered when a browser window or tab closes. This notifies Diffusion that the client is deliberately closing.

`diffusion-wrapper.js` can be found in `clients/flex` and `clients/silverlight` directories from a default installation.

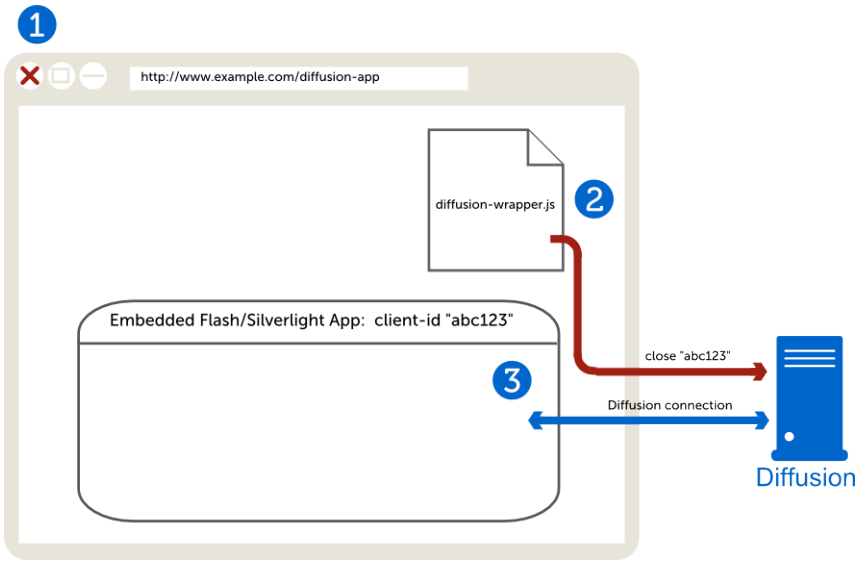


Figure 25: Diffusion wrapper

The preceding diagram shows `diffusion-wrapper.js` in use. When the user closes the containing browser tab or window, the following events occur:

1. The user closes the browser. This engages a JavaScript callback that calls `diffusion-wrapper.js`.
2. `diffusion-wrapper.js` runs and sends a closure request to the Diffusion server.
3. The Flash or Silverlight client dies. It has no chance to run cleanup code.

How to use Diffusion wrapper

The HTML page that loads the Flash/Silverlight app, must to load the Diffusion wrapper script.

The following example shows the `diffusion-wrapper.js` file being included in a web page:

```
<html>
 <head>
  <script src="diffusion-wrapper.js" language="javascript"></script>
 </head>
 <body>
 .....
 </body>
</html>
```

The Flash/Silverlight app makes a call to the external DiffusionWrapper method `setClientDetails`, typically in the `onConnection` callback, supplying the client ID and server URL. In the example below, the method `ExternalInterface.call("setClientDetails")` provides DiffusionWrapper with the client ID and server URL. DiffusionWrapper adds a method to the `onbeforeunload` event of the window, which informs the Diffusion server that the client is intentionally closing the connection when the browser window or tab is closed.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
                width="371" height="334" minWidth="955"
                minHeight="600" layout="absolute">
<mx:Script>
 import com.pushtechnology.diffusion.ConnectionDetails;
 import com.pushtechnology.diffusion.DiffusionClient;
 import com.pushtechnology.diffusion.ServerDetails;
 import com.pushtechnology.diffusion.events.DiffusionConnectionEvent;
 import com.pushtechnology.diffusion.events.DiffusionExceptionEvent;
 import com.pushtechnology.diffusion.events.DiffusionMessageEvent;
```

```
 private var theServerUrl:String = "http://127.0.0.1:8080"
 private var theInitialTopic:String = "Echo"
 private var theClient:DiffusionClient;

 private function onConnect():void{
     theClient = new DiffusionClient();
     var theConnectionDetails:ConnectionDetails = new
 ConnectionDetails(new ServerDetails(theServerUrl),null);

     // Add the listeners
     theClient.addEventListener(DiffusionConnectionEvent.CONNECTION,
 onConnection);
     theClient.addEventListener(DiffusionMessageEvent.MESSAGE,
 onMessages);
     theClient.addEventListener(DiffusionExceptionEvent.EXCEPTION,
 onException);

     //Lets Go...
     theClient.connect(theConnectionDetails);
 }

 private function onConnection(event:DiffusionConnectionEvent):void{
     // Is the client connected?
     if(event.isConnected()){
         //Check the externalInterface availability
        if(ExternalInterface.available){
          // Send the ClientId and serverURL to the DiffusionWrapper
    ExternalInterface.call("setClientDetails", theClient.getClientID(),
 event.getServerDetails().getURL());
         }else{
         //The externalInterface is not available, so the DiffusionWrapper
 is not called.
         }
   }
 }

 private function onMessages(event:DiffusionMessageEvent):void{
  //Message received
     theMessages.text += event.toString();
 }

 private function onException(event:DiffusionExceptionEvent):void{

  //Exception received
  theMessages.text += event.toString();
 }

</mx:Script>
 <mx:TextArea id="theMessages" x="18" y="40" width="335" height="283"
 text="Messages"/>
 <mx:Button id="bConnect" x="105.5" y="10" width="160" label="Connect"
 click="onConnect()"/>
</mx:Application>
```

Chapter 13

System management

In this section:

- Going to production
- General management
- Classic deployment
- Hot deployment
- Using JMX
- Statistics
- Diffusion monitoring console
- Basic integration with Splunk

This section discusses how to manage a Diffusion server and system as a whole.

Going to production

When going to production with Diffusion review the information in this section.

Before going to production

Security

Develop the handlers that you require secure your Diffusion solution. You can use handlers in the Diffusion server to manage authentication and authorization.

For more information, see User access control on page 415.

Testing

Thoroughly test your Diffusion solution before deploying it in production.

Diffusion provides a number of tools that you can use to test your solution. For more information, see Testing on page 575.

Performance

Understand how your Diffusion solution performs.

You can use the benchmarking suite (available on GitHub) to gauge the performance of your Diffusion solution. For more information, see Benchmarking suite on page 595.

You can view metrics and statistics for your Diffusion solution through the Publisher API or through MBeans. For more information, see Statistics on page 400.

Tuning

Tune your Diffusion solution to ensure optimum performance for your expected load and usage patterns.

For more information, see Tuning on page 496.

Putting your Diffusion solution into production

Diffusion and your network

- Load balancers on page 64
- Network security on page 432

In production

Monitoring

Monitor your Diffusion solution using the monitoring console.

For more information, see Diffusion monitoring console on page 402.

Diagnostics

If an issue occurs in production, you can use log out put and other diagnostics to understand and fix the issue.

For more information, see Diagnostics on page 513.

General management

You can start the Diffusion server by using one of the provided scripts.

Procedure

The Diffusion server is started using the `diffusion.bat` (or `diffusion.sh` on UNIX platforms) script in the issued bin directory.

Configuration files are loaded by default from the `etc` directory. An optional properties directory can be specified as a parameter to the `diffusion` command. If a properties directory is specified, any properties files in the specified directory are loaded in preference to those in the `etc` directory. The server logs where it loads each property file from.

Classic deployment

Installing publishers into a stopped Diffusion instance.

The publishers that are started when Diffusion starts must be defined in the configuration file `etc/Publishers.xml`. Publishers that do not start with Diffusion can also be defined in the `etc/Publishers.xml` and these can be started later using JMX.

The publishers must be present on the classpath of the Diffusion server. The recommended way to do this is to compile the publisher source code with the `diffusion.jar` they run with on the classpath. Package the publisher class files into a JAR file. This JAR file must be deployed to the `ext/` directory of the Diffusion installation. The Diffusion server will search the `ext/` directory and load all the JAR files it finds.

Related Links

Defining publishers on page 79
How to define publishers that start with Diffusion.

Creating a Publisher class on page 88
A publisher is written by extending the abstract `Publisher` class (see Publisher API) and overriding any methods that must be implemented to achieve the functionality required by the publisher.

Hot deployment

Installing publishers into a running Diffusion instance.

In addition to starting publishers by defining them in `etc/Publishers.xml`, you can install them into an already running Diffusion instance by a process known as hot deployment. Publishers can also be undeployed and redeployed, providing they implement the `isStoppable` method, and it returns true. You can also deploy dependent JAR files, configuration files and associated web pages for a publisher. All artifacts required for deployment are packaged within a DAR file.

What's in a DAR file?

A DAR file contains the JAR file that contains the publisher Java files, one or more XML configuration files, a manifest file, and any other files required by your publisher.

Figure 26: Example folder structure inside a DAR file

The root folder name is the name of the publisher.

The `MANIFEST.MF` file contains an attribute, `Diffusion-Version`, which specifies the minimum version number of Diffusion on which this publisher runs. This prevents deployment of publishers to Diffusion instances which might not support features of the publisher or have different API signatures.

```
Manifest-Version: 1.0
Diffusion-Version: 5.1.0
```

The `etc` directory contains files which are normally found in a Diffusion installation's `etc` directory, but contain only information relating to the publisher being deployed. Files that affect the operation of Diffusion and have no relationship to the publisher are not loaded. Valid configuration files are:

`etc/Aliases.xml`
Include this file if there are associated HTML files.

`etc/Publishers.xml`
You must include this file.

`etc/SubscriptionValidationPolicy.xml`
Include this file if it is referenced from the `etc/Publishers.xml` file.

The `Publishers.xml` file has the same structure and the one in a Diffusion installation's `etc` directory. For more information, see Publishers.

For example:

```
<publishers>
        <publisher name="MyPublisher">

 <class>com.pushtechnology.diffusion.test.publisher.MyPublisher</class>
                <start>true</start>
                <enabled>true</enabled>
        </publisher>
</publishers>
```

Put all Java code required by your publisher in the `ext` folder. All files in this directory are placed on the classpath for the publisher. You can include any required third-party JAR files or resources in this folder.

Building a DAR file

You can use the Maven™ plugin *mvndar* to build and deploy your publisher DAR file. Alternatively, you can use an Ant build file to perform the same function.

Building a DAR file using mvndar (recommended)

The mvndar plugin for Maven is developed and maintained on GitHub: https://github.com/pushtechnology/mvndar.

To use mvndar, reference it in the `plugins` section of your `pom.xml` file.

```
<plugins>
      ...
      <plugin>
        <groupId>com.pushtechnology.tools</groupId>
        <artifactId>dar-maven-plugin</artifactId>
        <version>1.2</version>
        <extensions>true</extensions>
      </plugin>
      ...
    </plugins>
```

The plugin runs during the project's package phase and creates the DAR file in the target directory.

Building a DAR file using Apache Ant

The following example `build.xml` for Ant takes a publisher developed as an Eclipse™ project, packages it in a temporary directory, and copies it to the directory where the Ant script is executed:

```
<project name="MyPublisher" default="maketmpable">
    <property name="publisher.name" value="MyPublisher" />
    <property name="jar.name" value="${publisher.name}.jar" />
    <property name="diffusion.dir" value="." />
    <property name="dar.name" value="${publisher.name}.dar" />

    <target name="makejar">
        <jar jarfile="${jar.name}" includes="**/*.class" basedir="." />
    </target>

    <target name="maketmpable" depends="makejar">
        <tempfile property="temp.file" destDir="${java.io.tmpdir}"
 prefix="publisher-name"/>
        <copy todir="${temp.file}/${publisher.name}">
            <fileset dir=".">
                <include name="etc/**" />
                <include name="ext/**" />
                <include name="html/**" />
                <include name="META-INF/**" />
            </fileset>
        </copy>
        <copy todir="${temp.file}/${publisher.name}" file="${jar.name}" />
        <jar jarfile="${dar.name}" includes="${publisher.name}/**"
 basedir="${temp.file}" manifest="META-INF/MANIFEST.MF" />
        <delete dir="${temp.file}" />
    </target>
</project>
```

Related Links

Using Maven to build Java Diffusion applications on page 600

Apache Maven is a popular Java build tool and is well supported by Java IDEs. You can use Apache Maven to build your Diffusion applications.

Building publishers and other server application code with Maven on page 601
The Diffusion API for publishers and other server application code is not available in the Push Technology public Maven repository. To build server components, you must install the product locally and depend on `diffusion.jar` using a Maven system scope.

Deployment methods

There are two ways to deploy a DAR file: file copy or HTTP.

File copy

To use this method, copy your DAR file to the deployment directory on the file system. If you enable auto-deployment in the `Server.xml` configuration file, Diffusion periodically scans a directory for new or updated DAR files and deploys them. In the case of an updated DAR, the existing publisher is undeployed (if possible) before being redeployed.

HTTP

If the deploy web service is running, you can `POST` the DAR file over HTTP. For example, you can use command line tools such as curl to deploy the publisher:

```
curl --data-binary @MyPublisher.dar http://localhost:8080/deploy
```

> **Warning:** We recommend you use the HTTP method of deployment in your test environments only. If you enable the deploy web service in your production environment, you must take additional security measures to prevent unauthorized or malicious access to the web service URL. For example, by setting up restrictions in your firewall.

To enable deployment through HTTP, you must enable the web service in the `WebServer.xml` configuration file. For example, include the following XML:

```
<http-service name="deploy-service" debug="true”>
 <class>com.pushtechnology.diffusion.service.DeploymentService</class>
 <url-pattern>^/deploy.*</url-pattern>
 <max-inbound-request-size>128m</max-inbound-request-size>
</http-service>
```

Ensure that the HTTP connector is configured to have an input buffer large enough to contain the entire DAR file. You can configure this in the `Connectors.xml` configuration file.

Undeployment

For publishers deployed using the file copy method, you can delete the DAR file from the deployment directory and on the next scan the server undeploys the publisher. A DAR file can be undeployed only if all of the publishers it contains are stoppable. If a DAR file fails to be undeployed, any future modifications to it are ignored.

It is important that any files that the deployment process has extracted from the DAR are not deleted until the publisher has been successfully undeployed. Publishers can also be undeployed through JMX by invoking the undeploy operation on associated MBean, for example

```
localhost/Server/com.pushtechnology.diffusion - Publisher - MyPublisher -
 undeploy()
```

Using JMX

Diffusion can be managed using JMX (Java Management Extensions).

JMX Ports

Diffusion binds to the TCP ports configured in `etc/Management.xml` to listen for JMX clients such as JConsole and JVisualVM. The configuration properties include the host name, the RMI registry port, and the JMX discovery port. In the default configuration in `etc/Management.xml`, the two port numbers are 1099 and 1100.

If you are using a firewall solution that employs Network Address Translation (NAT), you might still be unable to connect to Diffusion even when TCP ports 1099 and 1100 are left open. This is because of a well-understood weakness in the default protocol used by JMX: the server redirects the client's initial connection to a second host and port, problematically it employs the IP address of the server's NIC, rather than the IP address the client connected to the first port on.

The more popular and more secure solution is to establish an SSH connection to the firewalled Diffusion server, tunneling ports 1099 and 1100 through SSH, then using JMX to connect to the local ends of the tunneled ports.

If you are running on JDK 7u4 or later, the two ports can be set to the same value. This can simplify firewall configuration.

Related Links

Management on page 472
Management.xml - specifies system management properties.

MBeans

Diffusion registers MBeans for many of its principal features with the JMX service.

Annotations on each of the MBeans employed are used to produce the following pages in this manual as well as feeding JMX clients with descriptive information. MBeans, attributes and operations have descriptions; operation arguments have names; operations also have JMX impact information.

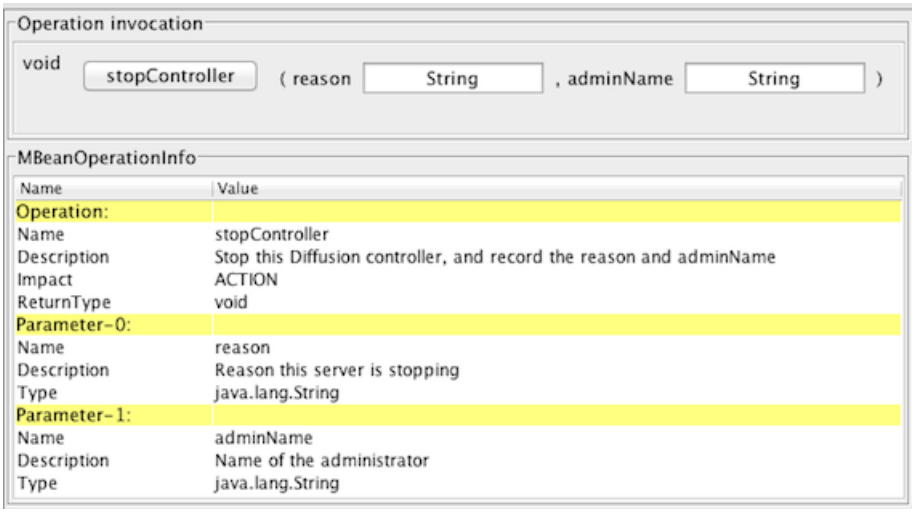


Figure 27: The server MBean stopController operation showing in JConsole

AggregateStatisticsMBean

Interface for a specific StatisticsCollector

Attributes

Name	Type	Read/Write	Description
instanceStatisticsEnabled	boolean	read-write	Whether statistics collection is enabled or not.
managementName	String	read	
managementType	String	read	

AuthorisationManagerInstanceMBean

Management interface to the optional AuthorisationManager

Attributes

Name	Type	Read/Write	Description
connections	int	read	Number of connections authorised
connectionsDenied	int	read	Number of connections denied authorisation
fetches	int	read	Number of fetches authorised
fetchesDenied	int	read	Number of fetches denied authorisation
handlerClassName	String	read	Class name of any configured AuthorisationHandler
hasHandler	boolean	read	true if this server has an AuthorisationHandler configured
subscriptions	int	read	Number of subscriptions authorised
subscriptionsDenied	int	read	Number of subscriptions denied authorisation
writes	int	read	Number of writes authorised
writesDenied	int	read	Number of writes denied authorisation

Notifications

Class Name	Description
javax.management.Notification	News that a client interaction with a topic was disallowed

BaseThreadPoolMBean

Management interface to basic thread pool

Attributes

Name	Type	Read/Write	Description
activeCount	int	read	The number of active threads
coreSize	int	read-write	The number of threads to maintain
keepAlive	long	read-write	keep alive time in milliseconds

Name	Type	Read/Write	Description
largestSize	int	read	the largest number of threads that have simultaneously been in the pool
maximumSize	int	read-write	Maximum Pool Size
queueLowerThreshold	int	read	The lower queue size at which the notification handler will be notified
queueMaximumSize	int	read	The maximum queue size that the task queue can reach
queueSize	int	read	The size of the embedded task queue
queueUpperThreshold	int	read	The upper queue size at which the notification handler will be notified
size	int	read	The current number of threads in the pool
taskCount	long	read	The approximate total number of tasks that have ever been scheduled for execution

ClientConnectionStatisticsMXBean

Monitoring interface to the client session statistics MBean

Attributes

Name	Type	Read/Write	Description
clientOutputFrequency	long	read-write	statistics output frequency in milliseconds
clientResetFrequency	long	read-write	the frequency at which the counters are reset
concurrentClientCount	int	read	The current client session count
connectionCounts	Map	read	The current client session count, broken down by client type
maximumConcurrentClientCount	int	read	The maximum number of concurrent client sessions
maximumDailyClientCount	int	read	The count of client sessions started in a day

ConnectorManageableMBean

Management interface to a Connector

Attributes

Name	Type	Read/Write	Description
keepAliveQueueMaximumDepth	int	read-write	the maximum queue depth used for clients in the keep alive state
keepAliveTime	long	read-write	the time in milliseconds that a unexpectedly disconnected client should be kept alive before closing
numberOfAcceptors	int	read	the number of Acceptors

Name	Type	Read/Write	Description
queueDefinition	String	read-write	the queue definition
totalNumberOfConnections	long	read	number of connections accepted since the Connector was started
uptime	String	read	a pretty printed string of the time this connector has been running, or 0 if not
uptimeMillis	long	read	the milliseconds this connector has been running, or 0 if not

Operations

Name	Return Type	Arguments	Impact	Description
remove	void	0	ACTION	Remove the Connector. It will not be possible to restart the Connector again (until system restart).

Name	Return Type	Arguments	Impact	Description
start	void	0	ACTION	Start the connector.

Name	Return Type	Arguments	Impact	Description
stop	void	0	ACTION	Stop the connector. Allows it to be restarted.

DiffusionControllerMBean

Diffusion Controller

Attributes

Name	Type	Read/Write	Description
buildDate	String	read	The build date and time
freeMemory	long	read	The amount of free memory in the Java Virtual Machine
licenseExpiryDate	Date	read	The license expiry date
maxMemory	long	read	The total amount of memory in the Java virtual machine
numberOfTopics	long	read	The number of topics on this server
release	String	read	The Diffusion release string, e.g. 4.5.2_01
startDate	Date	read	Time at which this Diffusion server began
startDateMillis	long	read	The time at which this Diffusion server was started, as milliseconds since the epoch
timeZone	String	read	name of the time zone to which this Diffusion server belongs

Name	Type	Read/Write	Description
totalMemory	long	read	The total amount of memory in the Java virtual machine
uptime	String	read	The time that this controller has been running, e.g. "3 hours 4 minutes 23 seconds"
uptimeMillis	long	read	The time this controller has be running, expressed in milliseconds
usedPhysicalMemorySize	long	read	Free physical memory, expressed in bytes
usedSwapSpaceSize	long	read	Used swap space, expressed in bytes
userDirectory	String	read	The directory in which the Diffusion server was begun
userName	String	read	Name of the user to which the Diffusion server belongs

Operations

Name	Return Type	Arguments	Impact	Description
checkLicense	void	0	ACTION	Recheck the license being used

Name	Return Type	Arguments	Impact	Description
stopController	void	0	ACTION	Stop this Diffusion controller

Name	Return Type	Arguments	Impact	Description
stopController	void	2	ACTION	Stop this Diffusion controller, and record the reason and adminName

Argument name	Type	Description
reason	String	Reason this server is stopping
adminName	String	Name of the administrator

EventPublisherMBean

Management inferface for an Event Publisher

Attributes

Name	Type	Read/Write	Description
totalNumberOfMessages	Long	read	the total number of messages processed
upTimeMillis	long	read	the time in milliseconds that the service has been running
uptime	String	read	the pretty-printed time that the service has been running

Operations

Name	Return Type	Arguments	Impact	Description
close	void	0	ACTION	Close the connection to this Event Publisher

JMXAdapterMXBean

Management interface to the adapter that reflects MBean attributes and notifications as Diffusion Topics

Attributes

Name	Type	Read/Write	Description
updateFrequency	long	read	MBean attribute poll frequency, in milliseconds

LogFileMXBean

Management interface for Log Definition

Attributes

Name	Type	Read/Write	Description
description	LogDescription	read	the LogDescription for this log
filename	String	read	The fully qualified filename of this log
logLevel	String	read-write	The current level as a string

MultiplexerManagerMBean

Management interface to the multiplexer manager

Attributes

Name	Type	Read/Write	Description
numberOfMultiplexers	int	read	the number of multiplexers

MultiplexerMBean

Management interface to the multiplexer

Attributes

Name	Type	Read/Write	Description
latencyWarningTime	long	read	The latency threshold of this multiplexer, after which notifications shall be sent
name	String	read	The Multiplexer name
numberOfClients	int	read	The current number of clients assigned to multiplexer

Notifications

Class Name	Description
javax.management.Notification	Published in case of multiplexer latency

PublisherControllerMBean

Management interface for a publisher

Attributes

Name	Type	Read/Write	Description
inboundClientAverageMessageSize	long	read	The average size of a message received from clients
inboundClientNumberOfMessages	long	read	The count of messages received from clients
inboundEventPublisherAverageMessageSize	long	read	The average size of a message received from Event Publishers
inboundEventPublisherNumberOfMessages	long	read	The count of messages received from Event Publishers
logLevel	String	read-write	The log level for this publisher
numberOfTopics	int	read	The count of topics associated with this Publisher
outboundAverageMessageSize	long	read	The average size of a message sent to clients
outboundNumberOfMessages	long	read	The count of messages sent to clients
started	boolean	read	true if the publisher is started

Operations

Name	Return Type	Arguments	Impact	Description
removePublisher	void	0	ACTION	Permanently remove the publisher

Name	Return Type	Arguments	Impact	Description
startPublisher	void	0	ACTION	Start this publisher

Name	Return Type	Arguments	Impact	Description
stopPublisher	void	0	ACTION	Stop this publisher

Name	Return Type	Arguments	Impact	Description
undeploy	void	0	ACTION	Un-deploy this publisher

StatisticsControllerMBean

Interface for StatisticsService controller

Attributes

Name	Type	Read/Write	Description
clientStatisticsEnabled	boolean	read-write	See if aggregate statistics for Clients are enabled
enabled	boolean	read-write	See if the server-wide Statistics service is enabled
publisherStatisticsEnabled	boolean	read-write	See if aggregate statistics for Publishers are enabled
serverStatisticsEnabled	boolean	read-write	See if aggregate statistics for Event Publishers are enabled
topicStatisticsEnabled	boolean	read-write	See if aggregate statistics for Topics are enabled
managementName	String	read	
managementType	String	read	

StatisticsManagerMBean

Monitoring interface to the collection of server statistics

Attributes

Name	Type	Read/Write	Description
statisticsEnabled	boolean	read-write	Global statistics switch

VirtualHostMBean

HTTP Virtual Host management interface

Attributes

Name	Type	Read/Write	Description
cacheSizeBytes	int	read	the cache size in bytes
cacheSizeEntries	int	read	the number of entries in the cache
aliasFile	String	read	the alias file name
aliasProcessor	HTTPAliasProcessor	read	the alias processor object
cache	HTTPCache	read	the HTTP cache
compressionThreshold	int	read	the compression threshold
debug	boolean	read	true if debug is set
documentRoot	String	read	the document root directory
errorPage	String	read	the error-page file name
fileServiceName	String	read	the file service name
homePage	String	read	the home-page file name
host	String	read	the host name

Name	Type	Read/Write	Description
minify	boolean	read	true if the minify property is set
name	String	read	the virtual host name
numberOfRequests	int	read	number of requests actioned since service was started
static	boolean	read	true if static
webServerName	String	read	the web server name

Operations

Name	Return Type	Arguments	Impact	Description
startService	void	0	ACTION	Restart a previously stopped virtual host

Name	Return Type	Arguments	Impact	Description
stopService	void	0	ACTION	Stops the virtual host from processing requests

Name	Return Type	Arguments	Impact	Description
clearCache	void	0	ACTION	clear the cache of all entries

Using Visual VM

VisualVM is a good tool for seeing what is happening inside the Diffusion instance.

Running Visual VM locally

Visual VM is sometimes installed with JDK's but can be downloaded from https://visualvm.dev.java.net/

Starting the instance of Visual VM locally presents a list of currently running Java processes.

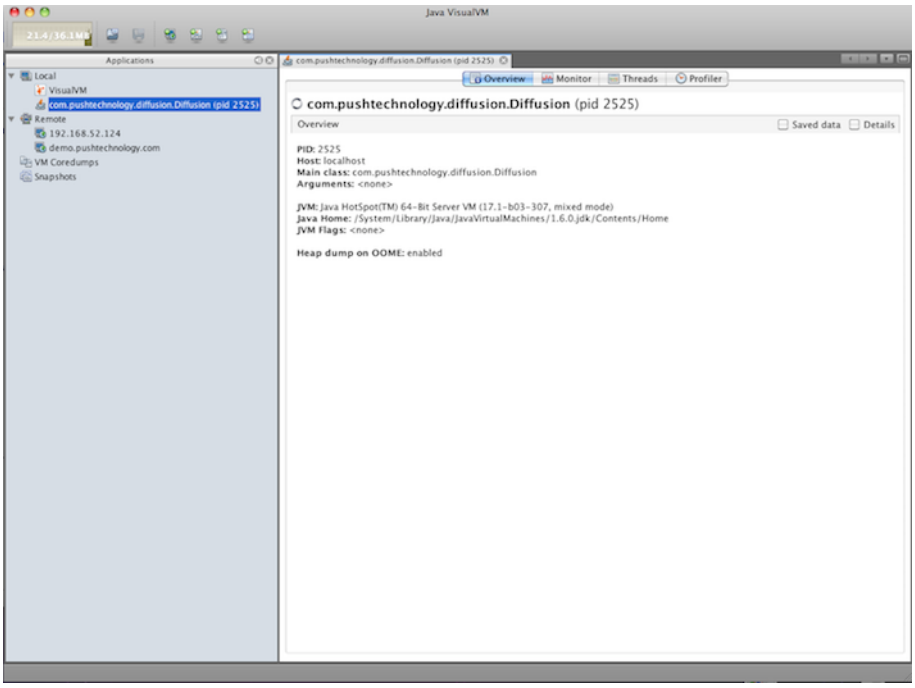


Figure 28: Java VisualVM: Overview tab

You can select multiple views (tabs) detailing the JVM itself.

Monitor

The monitor

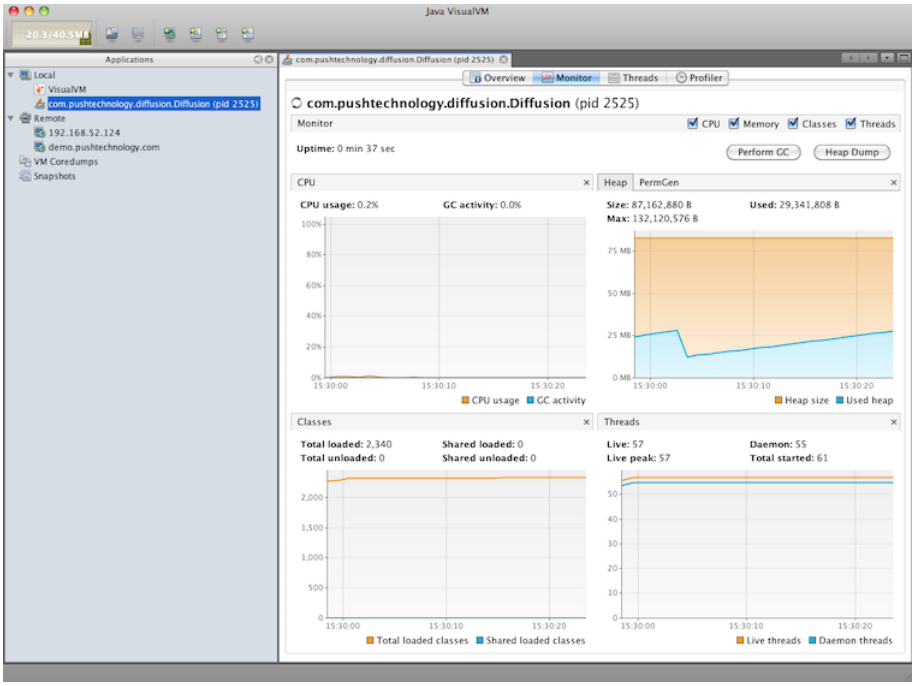


Figure 29: Java VisualVM: Monitor tab

Threads

The threads tab shows the number of live threads and daemon threads and related information.

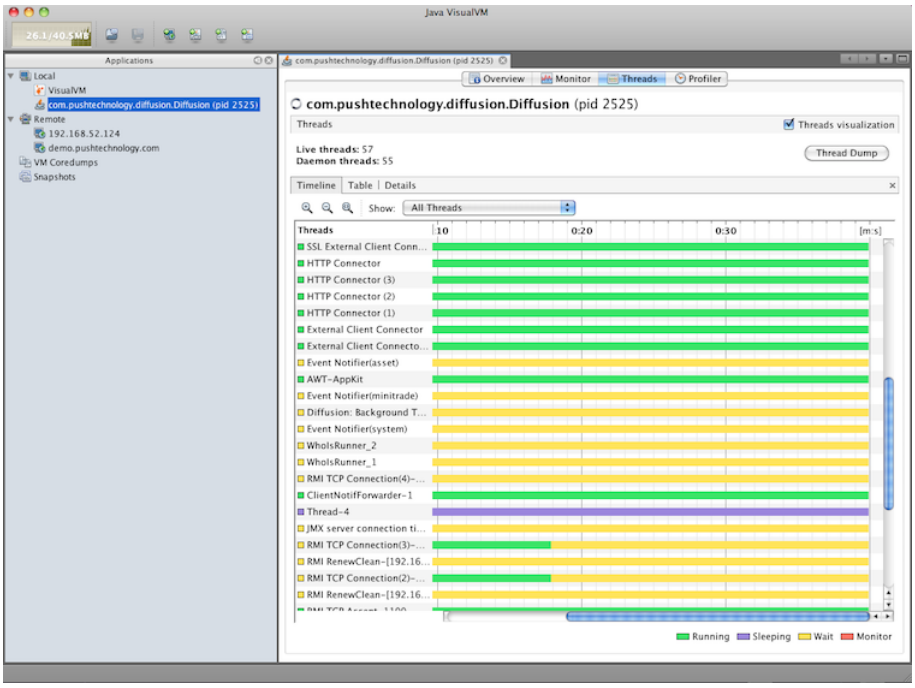


Figure 30: Java VisualVM: Threads tab

Profile

The profiler tab shows a table of profiled methods and related information.

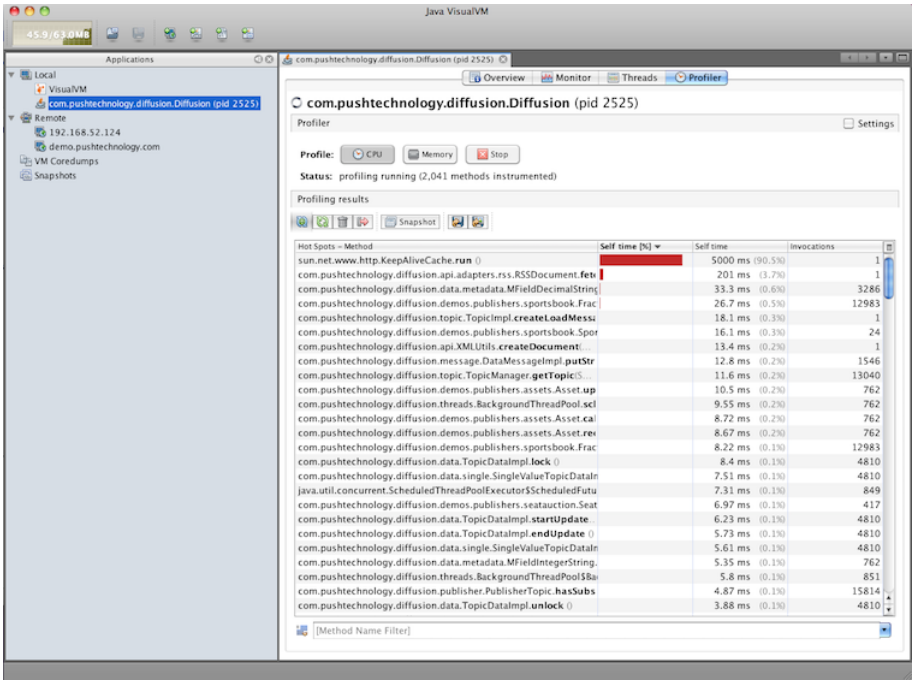


Figure 31: Java VisualVM: Profiler tab

Running Visual VM remotely

To run Visual VM remotely, you must run a process called Jstatd on the server. You might also be required to run the Jstatd process as the same user that owns the Diffusion process. To aid in this there is a jstatd(.sh/bat) shipped in the tools directory.

If you do not have the tools folder on the remote machine, you must start jstatd manually. To do this there must be a policy file in place. The policy file must contain the following text:

```
grant codebase "file:${java.home}/../lib/tools.jar" {
   permission java.security.AllPermission;
};
```

Add the following to the jstatd command.

```
-Djava.security.policy=jstatd.policy
```

When attaching to a remote instance it might be necessary to alter the Diffusion startup parameters to something like the following parameters:

```
-Dcom.sun.management.jmxremote.port=3333
-Dcom.sun.management.jmxremote.host={host}
-Djava.rmi.server.hostname={host}
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false -Dcom.sun.management.jmxremote
```

Note here that you can attach VisualVm to port 3333 and the normal JMX connection for Diffusion is 1099 (or whatever is set in `etc/Management.xml`)

As with JConsole visual VM can fail silently and not gave any reasons for not connecting. If you do have difficulty in connecting, adding this extra argument both client and server can help diagnose issues.

```
-Djava.rmi.server.logCalls=true
```

Using JConsole

Diffusion can be managed using the JMX system management console JConsole.

The properties specify a host name an RMI port and a connection port on which listens for JMX connections.

An admin user can be configured as required. Read-only users can be configured that can view but not manage the system in any way.

Connecting to the JMX service

The Diffusion server can be managed using a JMX system management console such as JConsole.

Figure 32: JConsole New Connection dialog: Local Process

Once connected to JMX, several aspects of the system are available to monitor and tune.

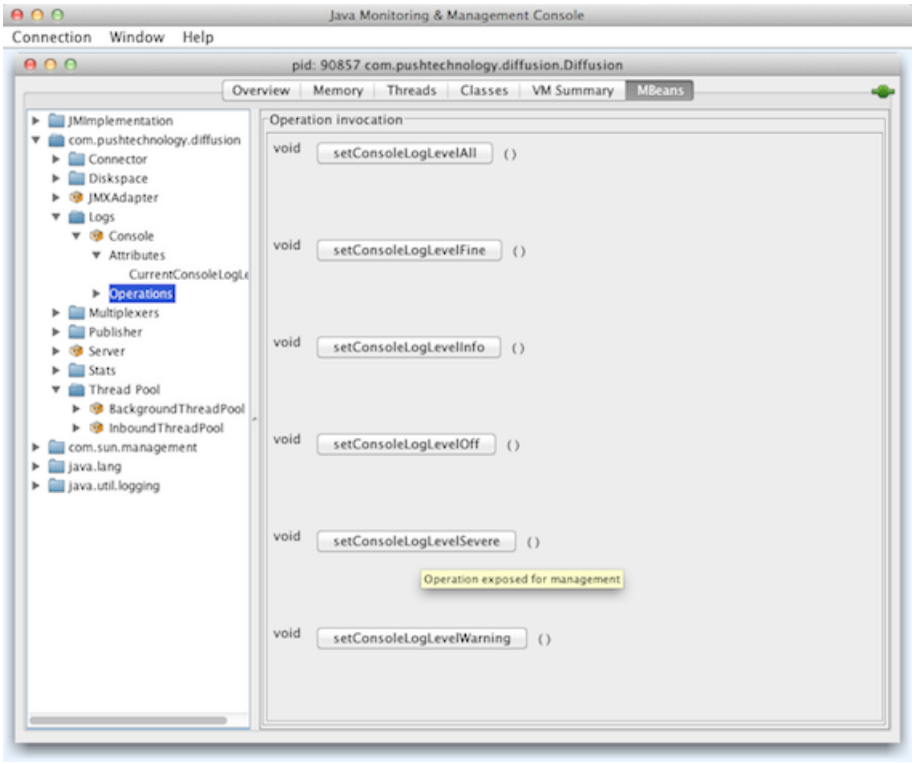


Figure 33: Tuning and monitoring in JConsole

This enables many views (tabs) on what is going on inside the JVM.

JConsole can be quite problematic on remote connections, and it fails silently. There are additional command line options that you can specify get more logging on the connection. Adding `-J-Djava.util.logging.config.file=logging.properties` to the jconsole command line. The logging.properties file must be created with the following entries:

```
handlers= java.util.logging.ConsoleHandler.level=INFO

java.util.logging.FileHandler.pattern = %h/java%u.log
java.util.logging.FileHandler.limit = 50000
java.util.logging.FileHandler.count = 1
java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter

java.util.logging.ConsoleHandler.level = FINEST
java.util.logging.ConsoleHandler.formatter =
 java.util.logging.SimpleFormatter

// Use FINER or FINEST for javax.management.remote.level - FINEST is
// very verbose...
//
javax.management.level=FINEST
javax.management.remote.level=FINER
```

Often the remote end (Diffusion server in this case) gets confused about what its remote server address is and you might have to add `-Djava.rmi.server.hostname={host or IP address}` to the Diffusion starting arguments.

Connecting to Diffusion on a remote server

Diffusion enables users to monitor and manage a remote Diffusion server with JConsole. The default configuration in `etc/Management.xml` establishes a listener on port 1099 for all network interfaces. Consequently jconsole users input the host and port into JConsole's **Remote Process** control. The username and password in question are also held in `etc/Management.xml`

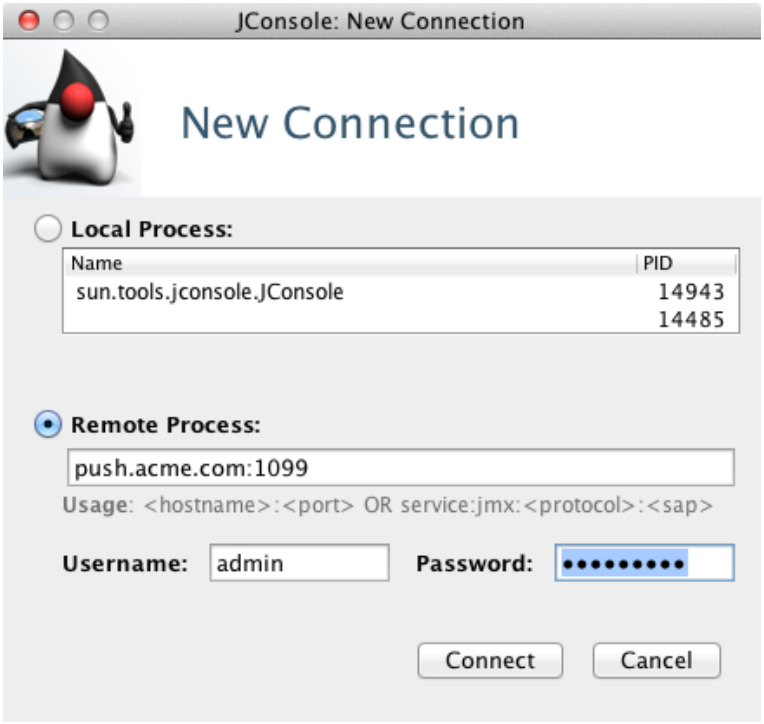


Figure 34: JConsole New Connection dialog: Remote Process

Statistics

Diffusion provides statistics about the server, publishers, clients, and topics.

Statistics in MBeans

Diffusion provides a set of statistics as MBeans. For more information about these statistics, see MBeans on page 386.

These statistics can be accessed in the following ways:

- Using a JMX tool, such as VisualVM or JConsole. For more information, see Using Visual VM on page 394 or Using JConsole on page 397.
- Using the Diffusion monitoring console. For more information, see Diffusion monitoring console on page 402.
- Through topics under the topic `Diffusion/Metrics`.

You can configure whether these statistics are collected and how they are reported. For more information, see Configuring statistics on page 401.

Statistics in the Publisher API

You can get some statistics from the Publisher API. For more information, see the Java Unified API documentation.

The following statistics are provided through the Publisher API:

- PublisherStatistics
  - InboundClientMessageStatistics
    - AverageMessageSize
    - BytesOnWire
    - NumberOfMessages
  - DEPRECATED: InboundEventPublisherMessageStatistics
    - AverageMessageSize

        - BytesOnWire
        - NumberOfMessages
    - OutboundMessageStatistics
        - AverageMessageSize
        - BytesOnWire
        - NumberOfMessages
- ClientStatistics
    - InboundMessageStatistics
        - AverageMessageSize
        - BytesOnWire
        - NumberOfMessages
    - OutboundMessageStatistics
        - AverageMessageSize
        - BytesOnWire
        - NumberOfMessages
- TopicStatistics
    - InboundMessageStatistics
        - AverageMessageSize
        - BytesOnWire
        - NumberOfMessages
    - OutboundMessageStatistics
        - AverageMessageSize
        - BytesOnWire
        - NumberOfMessages
    - TotalNumberOfSubscribers

Configuring statistics

You can configure statistics using the `etc/Statistics.xml` configuration file or programmatically using the Classic API.

The `Statistics.xml` configuration file

Diffusion servers provide statistics which are available through the JMX MBeans or through topics under the topic `Diffusion/Metrics`. If statistics is enabled via `etc/Statistics.xml`, users can take measurements including the average message size, number of messages per topic per second, etc.

The statistics configuration has several distinct elements, that allow granular control over what is enabled on server start.

- `<statistics>`

  The top-level element. Setting the enabled property to false will disable all statistics for the server.
- `<topics>`, `<clients>`, `<publishers>`, and `<server>`

  Enabling these elements provides aggregate metrics for each given class. Each element also contains a `<monitor-instances>` element that dictates whether specific instances of the parent class are monitored. Instance metrics require that the parent element is enabled.

The collection of metrics is configured separately from the distribution thereof. Within the `Statistics.xml` configuration file, there is also a `<reporters>` element which contains definitions of available reporters, which expose metrics over different mediums.

Some reporters allow certain properties to be passed to them. For instance, the topics reporter allows the use of `<property name="interval">` to provide the desired update frequency in seconds. Details of valid properties is documented within the `etc/Statistics.xml` file itself.

Programmatic configuration

You can programmatically query and configure the recording and calculation of statistics for the classes:

- `com.pushtechnology.diffusion.api.publisher.Client`
- `com.pushtechnology.diffusion.api.topic.Topic`
- `com.pushtechnology.diffusion.api.publisher.Publisher`

Developers are able to query the state of each class, to discover whether it is recording statistics using method `isStatisticsEnabled()`, stop recording with `stopStatistics()` and start recording using `startStatistics()`. The relevant API documentation holds more detail on the subject.

It has been observed that in a system with significant numbers of clients enabling the client instance statistics can result in a performance impact of up to 20% reduction in maximum throughput achieved, which can inhibit the system from supporting further connections.

If your system has more than 20k clients (at any given time) per Diffusion instance we recommend the client instance statistics be turned off.

Similarly if your system supports very large number of topics turning on the topic instance statistics might result in a performance hit.

By default statistic collection is turned off for performance reasons, to enable statistic collection, set the statistics root attribute `enabled`in `Log.xml` to true.

Diffusion monitoring console

A web console for monitoring your Diffusion solution

About

The Diffusion Monitoring Console is an optional publisher, provided as `console.dar`. It is deployed by default, and can be undeployed in the same manner as any DAR file. It exists to give operational staff using a web browser accessible visibility over the operations of a Diffusion solution

To manage a Diffusion server and make changes to it, use JMX tools such as JConsole. Unless you have to stop the Diffusion server, and stop and restart a publisher.

Dependencies

The console depends on the Diffusion publisher to mirror JMX MBeans as topics. The console also makes use of the statistics controlled by `etc/Statistics.xml`

The live graphing feature mandates a web browser that supports Scalar Vector Graphics (SVG). Most modern web browser implement the features required by the Console however Internet Explorer v9 is the recommended minimum for Microsoft users.

There are also two configuration settings within the Diffusion publisher config within `etc/Publishers.xml`. These are:

- `console.control.server` – Enable the ability to stop the Diffusion server through the console.
- `console.control.publishers` – Enable the ability to stop a particular publisher through the console.

Both of these options are disabled by default.

Features: Dashboard

Dashboard panels

By default the console is deployed as part of Diffusion. It is available in a fresh installation at http://localhost:8080/console.

Default layout

Started for the first time the console consists of six panels, each focusing on a key feature of the server

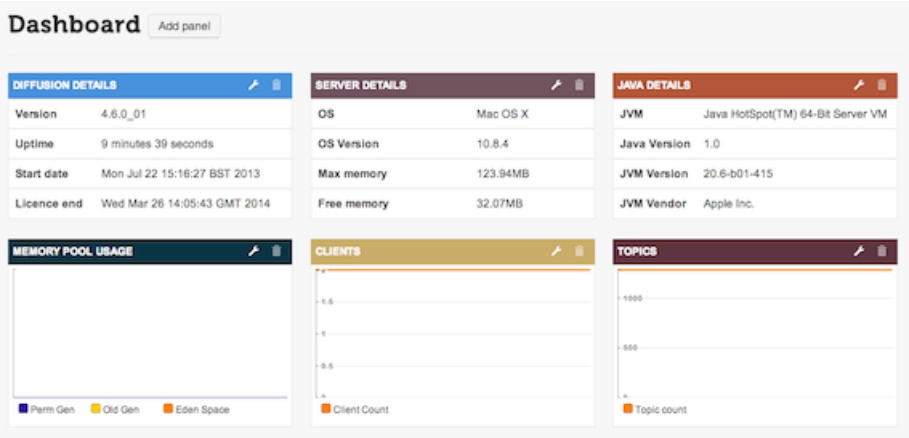


Figure 35: The default console layout

- **Diffusion Details**: the server version; the server up time, the server start date and time and the time and which the current license expires.
- **Server details**: the name and version of the underlying operating system; the total memory available (physical and virtual) and the amount of free memory.
- **Java Details**: the name, vendor and version of the Java Virtual Machine.

Instead of tabular data the second row show live line graphs.

- **Memory pool usage**: the values over time of the memory used by the Java VM process.
- **Clients**: the value over time of the number of Diffusion clients connected.
- **Topics**: the value over time of the number of topics on your Diffusion server.

Publishers table

At the bottom of the Dashboard is the publishers table. At a glance this shows the installed publishers and their vital statistics: the number of topics created, client connected, messages sent, bytes sent and finally publisher status

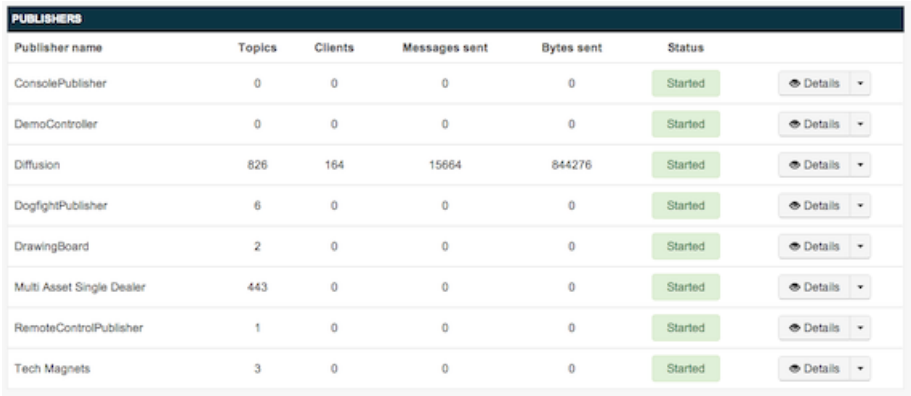

PUBLISHERS
Publisher name
Topics
Clients
Messages sent
Bytes sent
Status
ConsolePublisher 0 0 0 0 Started Details
DemoController 0 0 0 0 Started Details
Diffusion 826 164 15664 844276 Started Details
DogfightPublisher 6 0 0 0 Started Details
DrawingBoard 2 0 0 0 Started Details
Multi Asset Single Dealer 443 0 0 0 Started Details
RemoteControlPublisher 1 0 0 0 Started Details
Tech Magnets 3 0 0 0 Started Details

Figure 36: The table of publishers

Using the pull-down menu on the **Details** button publishers can be stopped and restarted. The **Details** button itself reveals the publisher statistics: clients, topics, average messages per second and average bytes per second.

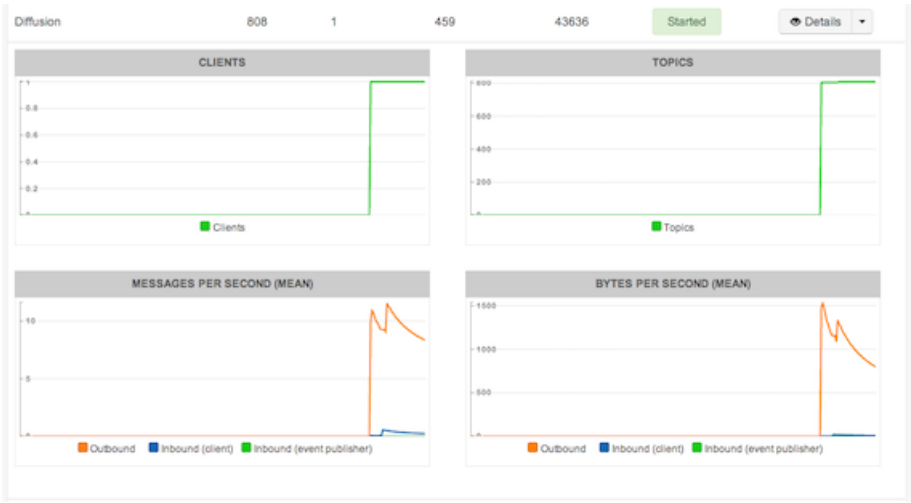


Figure 37: Publisher statistics graphs

Features: Topics tab

The Topics tab brings to the web browser the ability to browse and interact with the Diffusion topic tree.

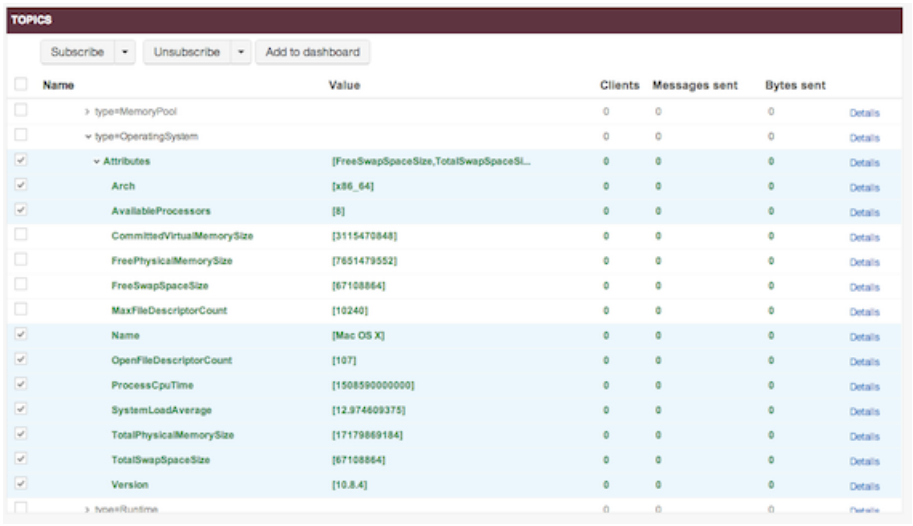


Figure 38: The table of topics

Users can intuitively browse the live topic tree, fetch and subscribe to topics. If the server is so configured the table also shows the number of subscribed clients, messages sent and bytes sent. Enable individual topic statistics through `etc/Statistics.xml`, for example,

```
<!-- Enable global topic statistics -->

  <topic-statistics enabled="true">

  <!-- Enable individual topic instance statistics -->

  <monitor-instances>true</monitor-instances>

</topic-statistics>
```

Once a set of topics is selected using its checkbox the Subscribe and Unsubscribe work intuitively, and each button has an recursive alternative available through the drop-down menu-button.

The details button shows more detail on the topic in question, as well as offering to fetch the topic value

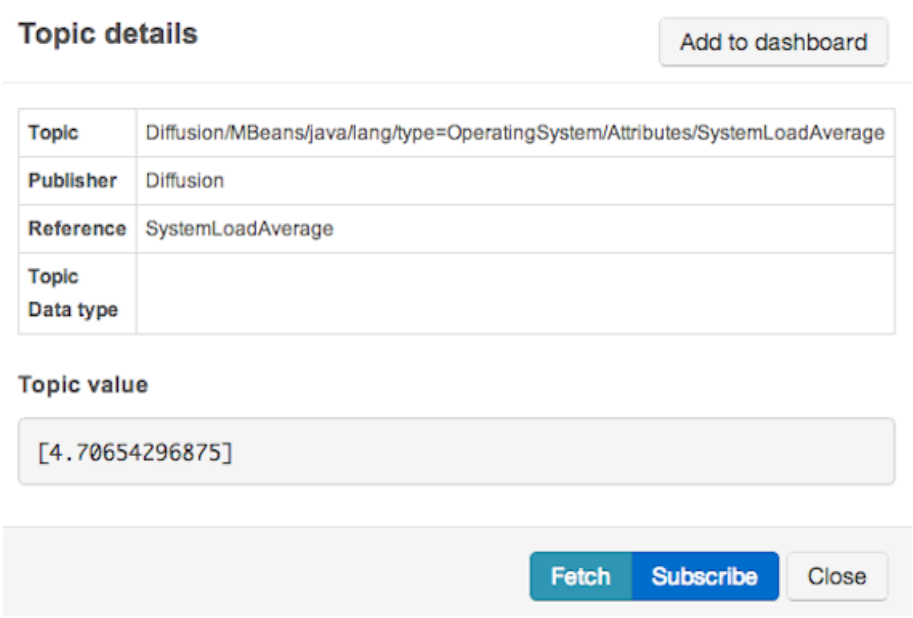


Figure 39: Details of the topic publishing the CPU load of the host server

Features: Clients tab

The Client tab shows a live list of the clients connected to the Diffusion server. Additionally it shows the number of messages to and from the server, the client IP address, connection type and connection time.

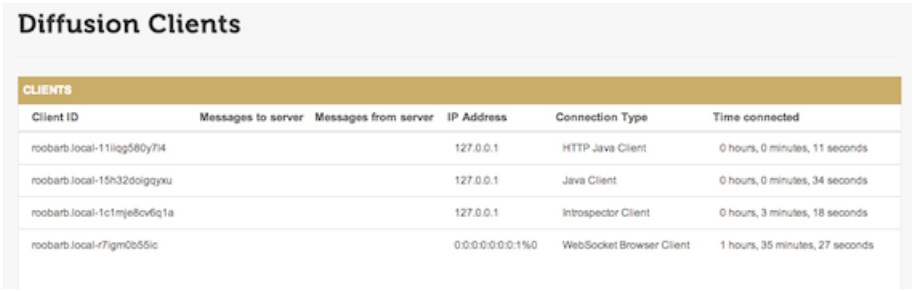


Figure 40: The table of clients

Configure the Diffusion server to provide live client statistics through `etc/Statistics.xml`

```
<!-- Enable global client statistics -->
<client-statistics enabled="true">
  <!-- Definition of the log in Logs.xml -->
  <log-name>stats</log-name>
  <!-- Specifies the output frequency of the log, this is one entry per
 frequency -->

  <output-frequency>1h</output-frequency>
  <!-- Enable individual client instance statistics -->
  <monitor-instances>true</monitor-instances>
</client-statistics>
```

Features: Logs tab

The Logs tab shows a live color-coded display of log entries emitted by the server up the level of **INFO**.

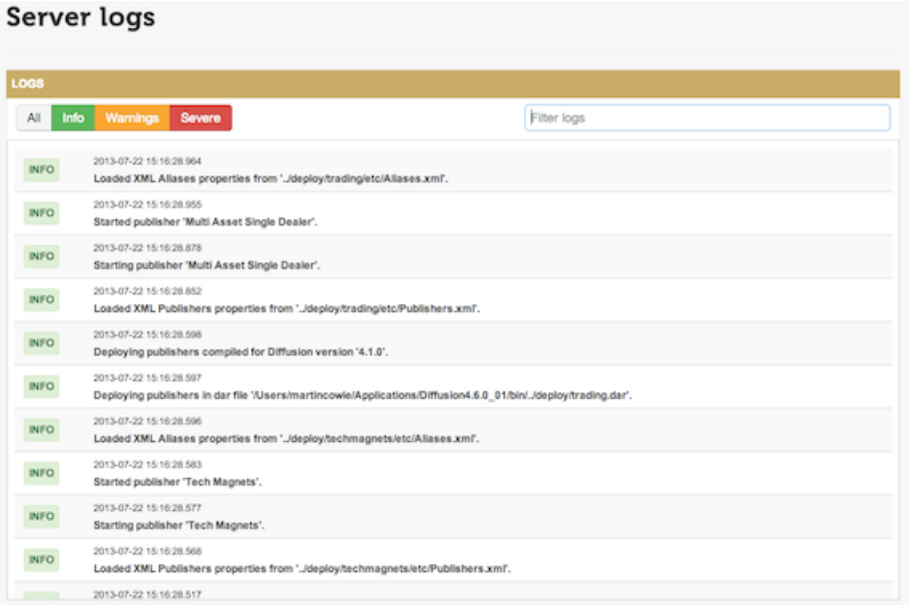


Figure 41: The table of log entries

Users can also perform client-side simple filtering on log entries. Unlike other monitoring metrics the Diffusion server retains up to 250 log entries in memory.

Advanced

Saving the console layout

Users can save changes made to their console layout with the “Save Dashboard layout” button. This persists a file on the server side, making it shared amongst all Console users.

White & Blacklist editing

The Console optionally maintains a blacklist or whitelist of IP addresses that are allowed to make use of it. Users can specify discrete IP addresses or use syntax supported by `etc/SubscriptionValidationPolicy.xml` to cover subnets. In order to make these changes active, after editing the whitelist or blacklist and clicking the "Save settings" button, you must restart the server.

Figure 42: Editing the Access Policy

Stop Diffusion

The **Stop Diffusion** menu item stops the server when clicked upon.

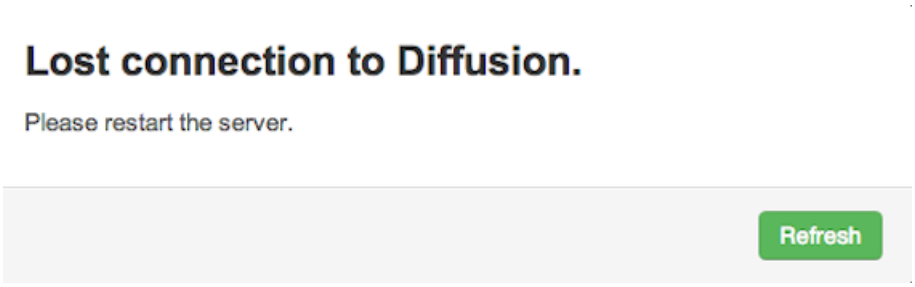


Figure 43: Notification that the Diffusion server has stopped

Going further

Changing the console layout

The console is designed to be extensible and flexible. Users can reorder, edit, create and remove panels. Grab the panel header and drag it to a new location as desired. Click the trash-can icon to remove the icon – with verification

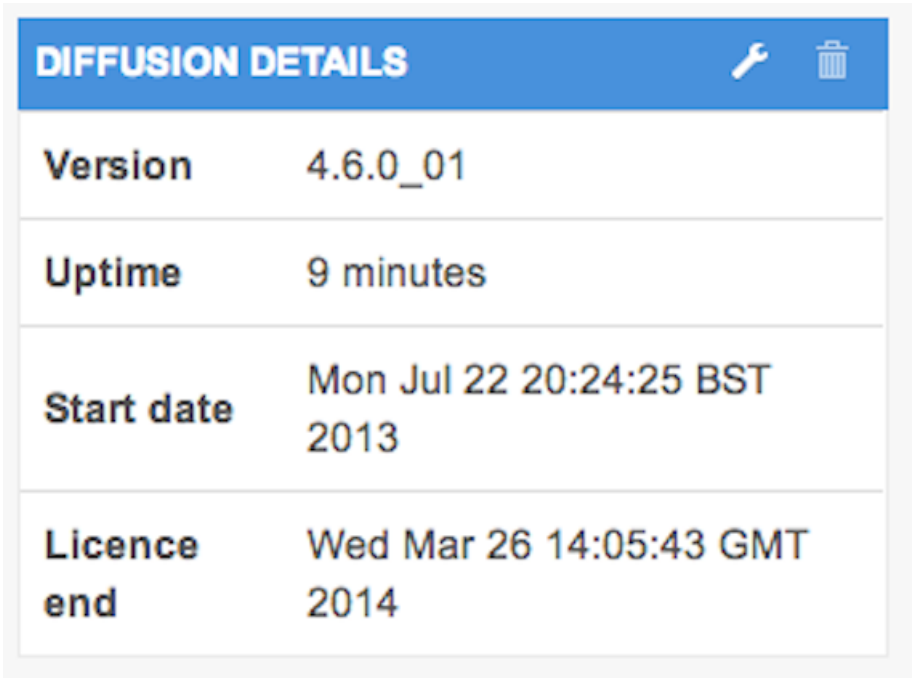


Figure 44: The default Diffusion Details panel

Click on the spanner or wrench icon to configure the panel.

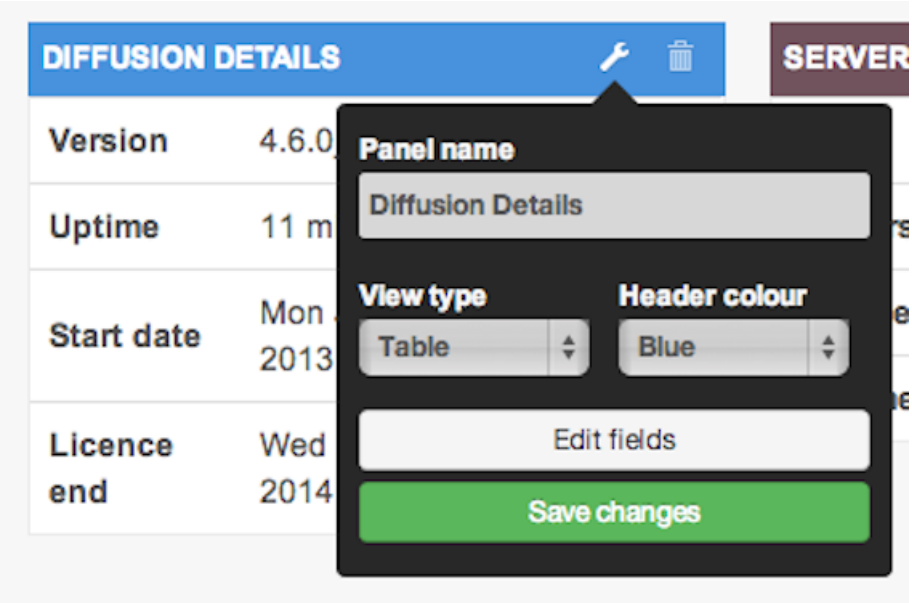


Figure 45: Editing the properties of the Diffusion Details panel

Panel name' and 'Header color' are self explanatory. 'View type' is a choice of data renderings.

- **Table**: As seen already, this option shows one or more monitoring metrics in a table of textual values.
- **Line**: Renders one or more numeric metrics as a line graph.
- **Area**: Renders one or more numeric metrics as an overlapping area graph.
- **Single**: Used to visualize a single metric in large text, for metrics that are worth the screen real-estate.

Line and area graphs have an extra two configuration fields: 'Refresh rate (ms)' and 'Max data points'. The latter configures how much data is retained for rendering the graph. The former governs the frequency with which the graph is updated and does not influence the frequency of updates from the server. Historic data is only stored in the browser and refreshing the page loses the stored set of data points.

Hovering the mouse over a graph panel shows the detail of the underlying data point

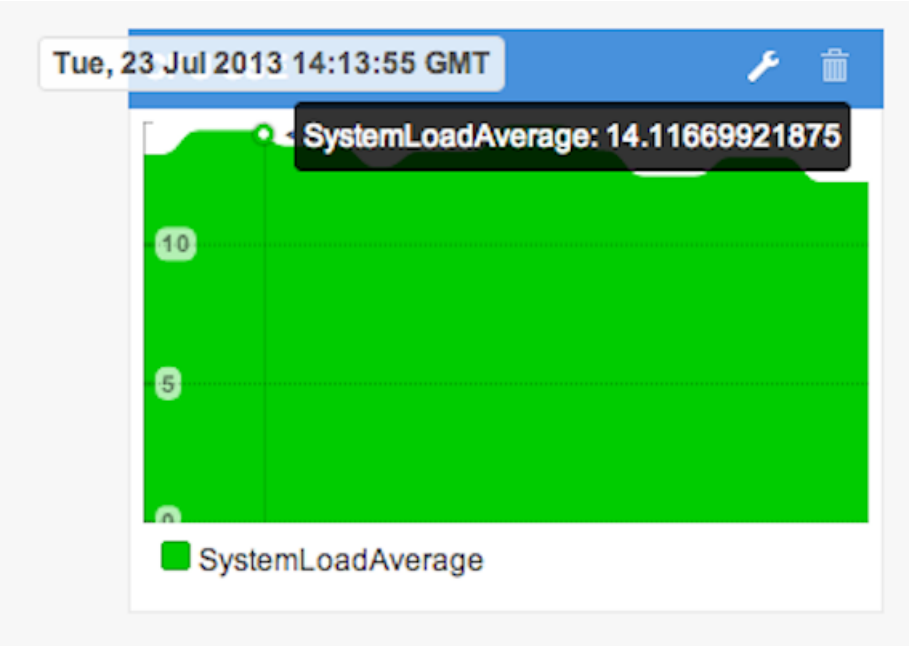


Figure 46: Visualizing the CPU load on a server at a specific time.

Sourcing monitoring metrics

Clicking the 'Edit fields' button presents the user with a Topic Data Fields dialog, where the user nominates one or more topics from where metrics are drawn.

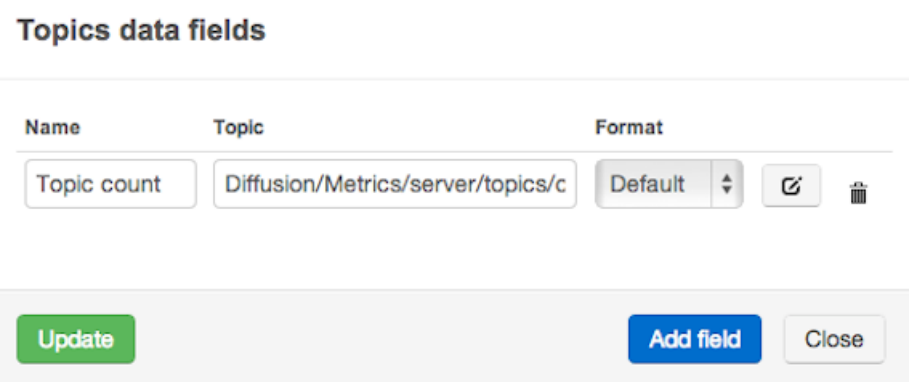


Figure 47: Editing and adding to the set of topics for this panel

Users of the **Topics** tab have already seen the **Add to dashboard** button in the **Topic details** dialog that can shortcut this process.

The default Console layout draws metrics from topics in the Diffusion/MBeans topic tree, however this is not mandatory and solution implementers are free to draw on any suitable topic to reflect their own monitoring needs – including 3rd party topics implemented as part of the solution.

The `Diffusion/MBeans` topic tree is populated by the JMX adapter which reflects all JMX MBeans as topics. Solution implementers that build custom MBeans to manage their solution can re-use the same MBeans for monitoring purposes.

The Console can draw on features that are themselves optional (Topic and client statistics, for example). If they are disabled, the Console points this out, and request they be enabled in `etc/Statistics.xml`

Production deployment notes

Securing the `Diffusion/` topics

The topics in the `Diffusion/` tree convey a great deal of power and it is highly probable that bringing a Diffusion based solution to production requires limiting their access to suitable users: for example, users with an IP address in a specific range. Solution implementers can achieve this by implementing an auth-handler.

The default configuration for the console allows users to stop and restart publishers as well as stop the Diffusion server itself. This feature is configured using the properties `console.control.server` and `console.control.publishers` on the Diffusion publisher in `etc/Publishers.xml`.

Basic integration with Splunk

How to achieve basic integration between Diffusion and the Splunk™ analysis and monitoring application

About

Splunk is a third-party application from Splunk, Inc., which provides monitoring and analysis of other applications, primarily by parsing their logs and extracting information of interest. The information is displayed through a web interface, which allows the creation of dashboards and alerts on user-defined events. Splunk is available for all major operating systems.

The Diffusion log format is designed to be consistent and to allow for easy parsing by monitoring tools, not limited to Splunk.

Installation

Installation typically takes just a few minutes, see the appropriate section of the Installation Manual. For simplicity, we assume that Diffusion and Splunk are installed on the same machine.

Basic configuration

This is easier to do with existing log files to import, so configure Diffusion to write log files. To better demonstrate Splunk, set the server log file to `FINEST` logging in `etc/Logs.xml` and start Diffusion.

```
<!-- Example server log configuration -->
<log name="server">
 <log-directory>../logs</log-directory>
 <file-pattern>%s.log</file-pattern>
 <level>FINEST</level>
 <xml-format>false</xml-format>
 <file-limit>0</file-limit>
 <file-append>false</file-append>
 <file-count>1</file-count>
 <rotate-daily>false</rotate-daily>
</log>
```

On startup, access the Splunk web UI at http://localhost:8000. After logging in (and changing the default admin password), choose the **Add data** option.

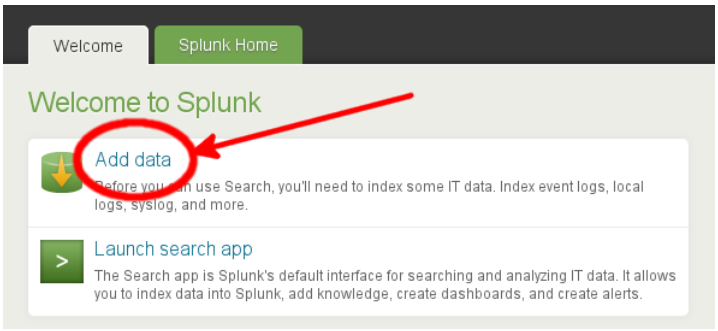
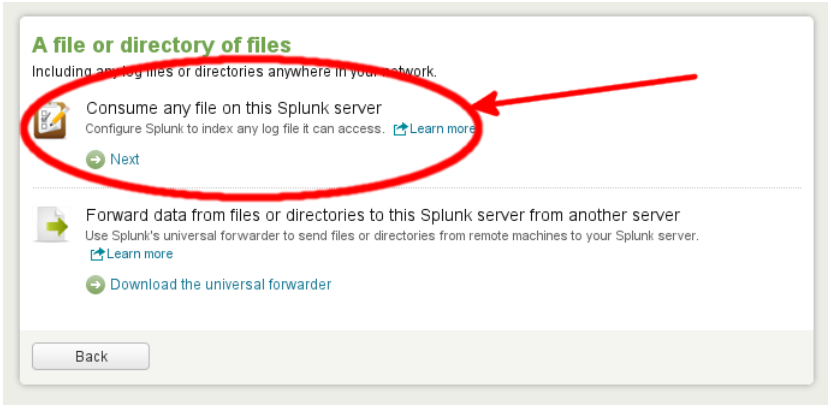

Welcome
Splunk Home
Welcome to Splunk
Add data
Launch search app
The Search app is Splunk's default interface for searching and analyzing IT data. It allows you to index data into Splunk, add knowledge, create dashboards, and create alerts.

Figure 48: Welcome tab of the Splunk web UI

In the **Add Data to Splunk** screen that follows, choose the link **A file or directory of files** followed by **Consume any file on this Splunk server**.


A file or directory of files
Consume any file on this Splunk server
Configure Splunk to index any log file it can access.
Next
Forward data from files or directories to this Splunk server from another server
Use Splunk's universal forwarder to send files or directories from remote machines to your Splunk server.
Learn more
Download the universal forwarder
Back

Splunk might not be able to immediately identify the format of the log files; if this is the case, a dialog box similar to the following is presented. Select **csv** from the existing source types. Diffusion uses a pipe symbol rather than a comma as a separator but this is acceptable to the Splunk CSV parser.

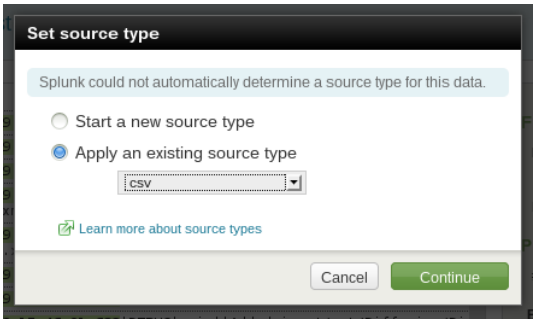

Set source type
Splunk could not automatically determine a source type for this data.
Start a new source type
Apply an existing source type
csv
Learn more about source types
Cancel
Continue

Figure 49: The Splunk Set source type dialog

The next dialog allows you to select the Diffusion `logs/Server.log` file under the **Preview data before indexing** option, which Splunk reads and parses. On the **Data Preview** screen, there are numbered log entries with the timestamp highlighted. This indicates that the log file has been correctly parsed. Accept this, and on the next screen, set the source to be continuously indexing the data. You can leave the parameters in **More settings** at their default values. Once this is done, you have given the new data source a name (for example, "Diffusion Server Log") and finally accepted the settings, you can begin searching and generating reports based on the log contents.

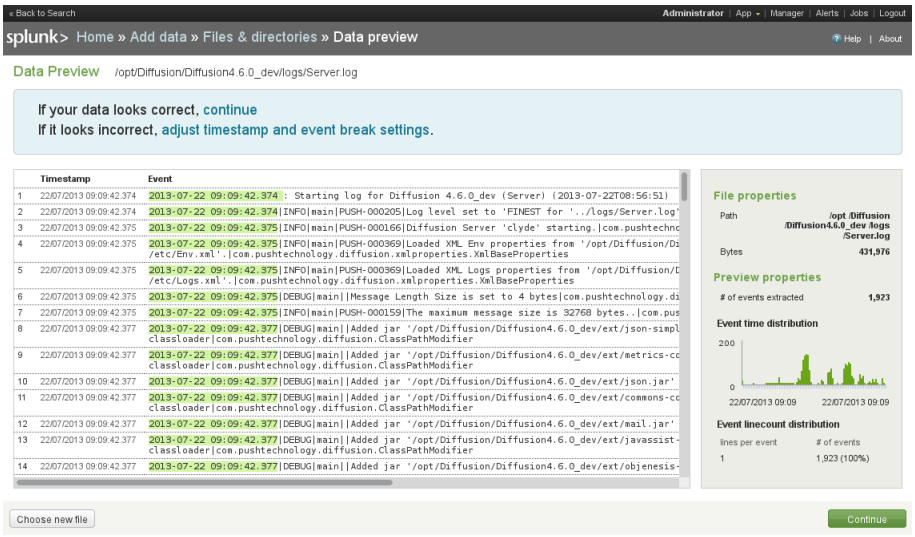

splunk> Home » Add data » Files & directories » Data preview
Data Preview /opt/Diffusion/Diffusion4.6.0_dev/logs/Server.log
If your data looks correct, continue
If it looks incorrect, adjust timestamp and event break settings.
File properties
Preview properties
Event time distribution
Event linecount distribution
Choose new file
Continue

Figure 50: The Data Preview panel

Simple searches

Now we have a data source configured, we can start to execute basic searches.

On the Splunk launch page, select the **Search** option. On the **Search Summary** page that opens, select the Source relating to the file `logs/Server.log` previously imported. The page changes to include the source in the Search area. Additional search terms can be added to the end, for example, “Started Publisher”.

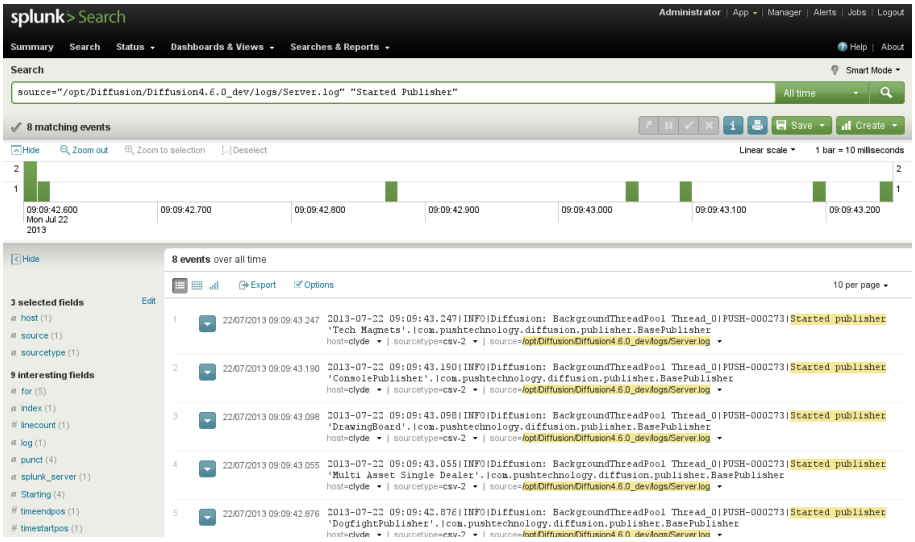

splunk> Search
Summary Search Status Dashboards & Views Searches & Reports
source="/opt/Diffusion/Diffusion4.6.0_dev/logs/Server.log" "Started Publisher"
8 matching events
8 events over all time

Figure 51: The Splunk search summary panel

Related Links

http://docs.splunk.com

Chapter 14

Security

In this section:

- User access control
- Network security

This section covers security issues within Diffusion and describes the facilities available within Diffusion relating to security.

User access control

Diffusion provides facilities for controlling client access to a Diffusion server.

You can implement authentication handlers to authenticate a client when it connects to the server based on any criteria you choose (for example, an LDAP look-up). For more information, see Authentication handlers on page 418.

You can implement authorization handlers to manage authorization and permissioning on actions that a client tries to take. For more information, see Authorization handlers on page 430

> **Note:**
>
> Previous versions of Diffusion used the authorization handlers to authenticate a client or user on connection to the server. This use of authorization handlers is now deprecated. For backwards compatibility, authorization handlers are still called for authentication operations. For more information, see Authentication on page 415.
>
> However, we recommend you implement your authentication logic using authentication handlers.

In addition, you can constrain client access by using client validation policies. For more information, see Client validation on page 95.

Related Links

Client validation on page 95
Diffusion provides the facility to validate client connections to determine whether they are allowed using whitelists (allowed clients) and blacklists (barred clients). This can be made to occur automatically or can be controlled programmatically from within publishers.

server on page 444
Server.xml - defines general Diffusion server properties as well as multiplexers, security, conflation, client queues, and thread pools.

Authentication

You can implement and register handlers to authenticate clients when the clients try to perform operations that require authentication.

The handlers you can implement and register are the following:

- Any number of local authentication handlers
- Any number of control authentication handlers
- At most one authorization handler

> **Note:** Using authorization handlers for authentication is deprecated.

The server calls the authentication handlers (local and control) in the order that they are defined in the `Server.xml` file. Then, if required, the server calls the authorization handler.

If no handlers are defined, the server allows the client operation by default.

Authentication process

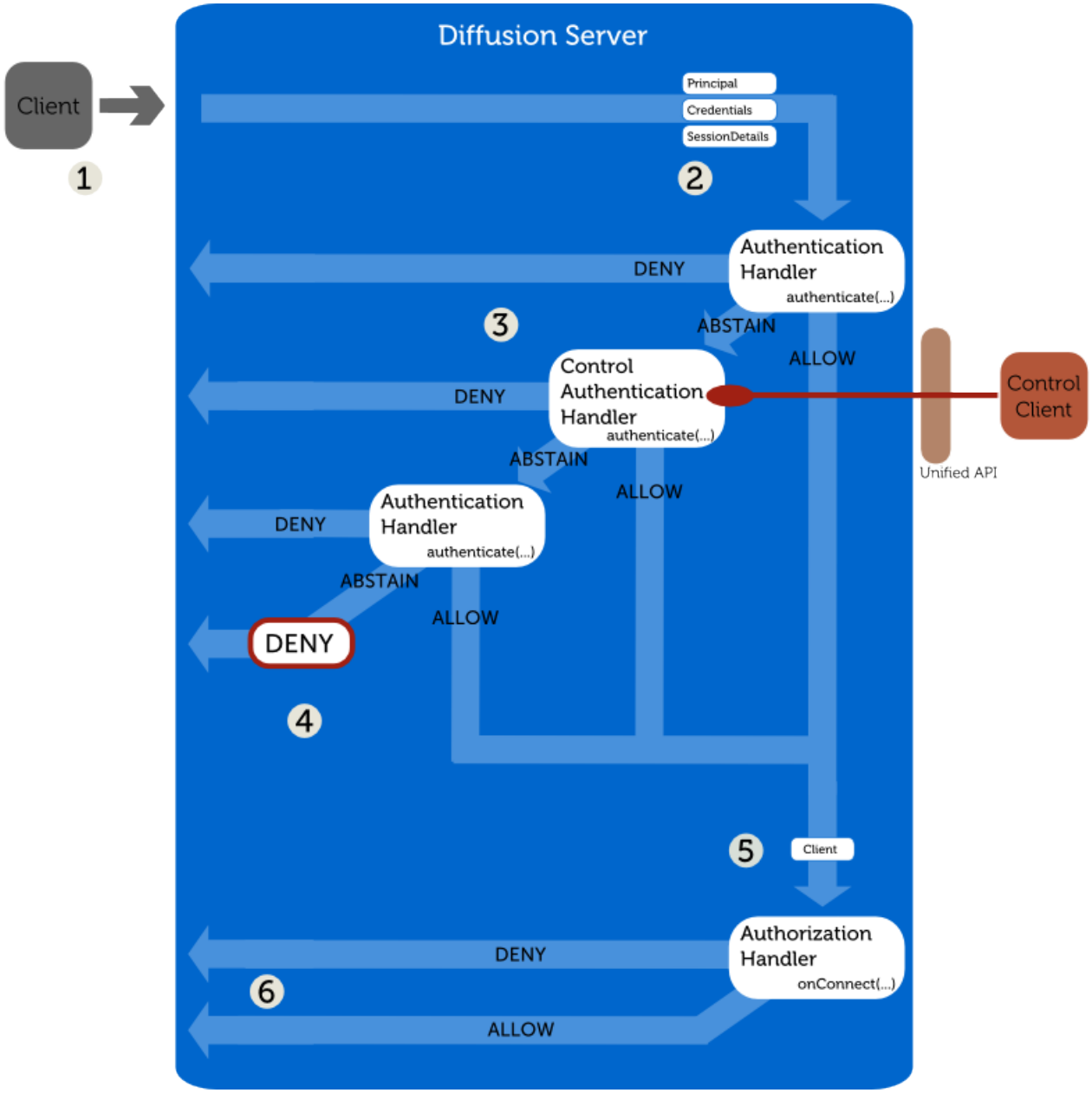


Figure 52: Authentication process for clients

1. A client tries to perform an operation that requires authentication. For more information, see Client operations that require authentication on page 417.
2. The server calls the authentication handlers one after another in the order that they are listed in the `Server.xml` file. It passes the following parameters to each authentication handler's `authenticate()` method:

   `Principal`

   A string that contains the name of the principal or identity that is connecting to the server or performing the action. This can have a value of `Session.ANONYMOUS`.

   `Credentials`

   The `Credentials` object contains an array of bytes that holds a piece of information that authenticates the principal. This can be empty or contain a password, a cryptographic key, an image, or any other piece of information. The authentication handler is responsible for interpreting the bytes.

   `SessionDetails`

This contains information about the client. The available details depend on what information the server holds about the client session. Some session information might not be available on initial connection.

This information can be used in the authentication decision. For example, an authentication handler can allow connection only from clients that connect from a specific country.

When it registers with the server, a control authentication handler can specify what details it requires, so only these details are sent by the server (if available). This reduces the amount of data sent across the control client connection.

`Callback`

A callback that the authentication handler can use to respond to the authentication request by using the callback's `allow()`, `deny()`, or `abstain()` method.

If the authentication handler is a local authentication handler, the authentication logic is done on the server. If the authentication handler is a control authentication handler, the parameters are passed to a control client and the control client handles the authentication logic and returns a response.

3. Each authentication handler can return a response of `ALLOW`, `DENY`, or `ABSTAIN`.
   - If the authentication handler returns `DENY`, the client operation is rejected.
   - If the authentication handler returns `ALLOW`, the decision is passed to the authorization handlers. If no authorization handlers are defined, the client operation is allowed.
   - If the authentication handler returns `ABSTAIN`, the decision is passed to the next authentication handler listed in the `Server.xml` configuration file.
4. If all authentication handlers respond with an `ABSTAIN` decision, the response defaults to `DENY`.
5. If an authorization handler is configured in the `Server.xml` file, the server calls it. It passes the following parameter to the authorization handler's `onConnect()` method:
   `Client`

   The `Client` object has an associated `Credentials` object. This `Credentials` object is part of the Classic API and is different to the `Credentials` object used by the authentication handlers. The Classic API `Credentials` object contains two strings: username and password. The username string is equivalent to the Principal string used by the Unified API.
6. The authorization handler can return a response of `ALLOW` or `DENY`.
   - If the authorization handler returns `DENY`, the client operation is rejected.
   - If the authorization handler returns `ALLOW`, the client operation is allowed.

Client operations that require authentication

The following client operations require authentication with the server:

Table 80: Client operations that require authentication

Client operation	API version	Behavior if operation is allowed	Behavior if operation is denied
Connect to server	Unified API	The client connection to the server is accepted.	The client connection to the server is rejected and is dropped.
Connect to server	Classic API	The client connection to the server is accepted.	The client connection to the server is rejected and is dropped.

Client operation	API version	Behavior if operation is allowed	Behavior if operation is denied
Change the principal associated with a client session	Unified API	The principal is changed.	The principal is not changed, but the client session is not dropped.
Change the principal associated with a client session	Classic API	The principal is changed. In the Classic API, the principal is the username string inside the `Credentials` object.	The principal is not changed, but the client session is not dropped.

Authentication handlers

You can implement authentication handlers that authenticate client connections to the Diffusion server or perform an action that requires authentication.

The authentication handlers can be implemented either remotely, in a control client, or locally, on the server. The authentication handlers can also be individual authentication handlers, that perform a single authentication check, or composite authentication handlers, that delegate to one or more individual authentication handlers.

> **Note:** Where there notation `c.p.d` is used in class or package names, it indicates `com.pushtechnology.diffusion`.

Local authentication handlers

A local authentication handler is an implementation of the `c.p.d.client.security.authentication.AuthenticationHandler` interface. Local authentication handlers can be implemented only in Java. The class file that contains a local authentication handler must be located on the classpath of the Diffusion server.

For more information, see Authentication API on page 428.

Control authentication handlers

A control authentication handler is an implementation of the `c.p.d.client.features.control.clients.AuthenticationControl.ControlAuthenticationHandl` interface in the Unified API.

A control authentication handler can be implemented in any language where the Diffusion Unified API includes the AuthenticationControl feature.

For more information, see AuthenticationControl on page 267.

Composite authentication handlers

A composite authentication handler delegates the authentication decision to an ordered list of one or more individual authentication handlers and returns a combined decision.

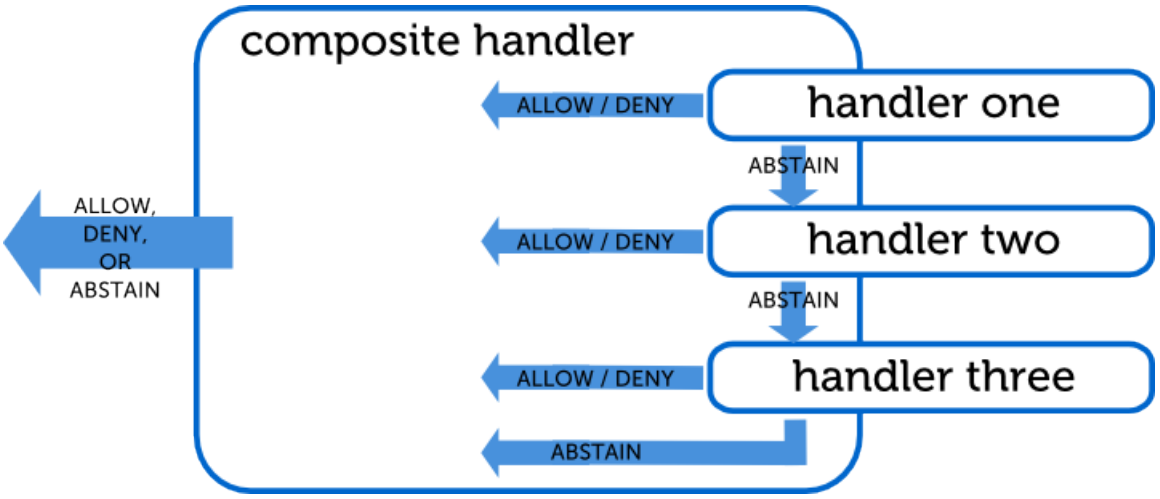


Figure 53: A composite authentication handler

- If an individual handler allows the client action, the composite handler responds with an allow decision.
- If an individual handler denies the client action, the composite handler responds with a deny decision.
- If an individual authentication handler abstains, the composite handler calls the next individual handler.
- If all individual handlers abstain, the composite handler responds with an abstain decision.

A composite authentication handler can be either local or control. A local composite authentication handler can delegate the authentication decision to one or more authentication handlers. A composite control authentication handler can delegate the authentication decision to one or more control authentication handlers.

The use of composite authentication handlers is optional. There are two reasons to consider using them:

- Composite authentication handlers enable you to combine authentication handlers together, which reduces the possibility of misconfiguration.
- Composite control authentication handlers improve efficiency by reducing the amount of number of messages sent between the server and control clients.

The following table matrix shows the four types of authentication handler.

Table 81: Types of authentication handler

	Individual	Composite
Local	Implement the `AuthenticationHandler` interface. For more information, see Developing a local authentication handler on page 421.	Extend the `CompositeAuthenticationHandler` class. For more information, see Developing a composite authentication handler on page 422
Control	Implement the `ControlAuthenticationHandler` interface. For more information, see Developing a control authentication handler on page 424.	Extend the `CompositeControlAuthenticationHandler` class. For more information, see Developing a composite control authentication handler on page 426

Related Links

Authentication API on page 428

The interfaces that are used for authentication are contained in the packages `com.pushtechnology.diffusion.client.security.authentication` and `com.pushtechnology.diffusion.client.types`.

AuthenticationControl on page 267
Use the AuthenticationControl feature to enable a control client session to authenticate other clients.

server on page 444
Server.xml - defines general Diffusion server properties as well as multiplexers, security, conflation, client queues, and thread pools.

Developing a local authentication handler on page 421
Implement the `AuthenticationHandler` interface to create a local authentication handler.

Developing a composite authentication handler on page 422
Extend the `CompositeAuthenticationHandler` class to combine the decisions from multiple authentication handlers.

Developing a control authentication handler on page 424
Implement the `ControlAuthenticationHandler` interface to create a control authentication handler.

Developing a composite control authentication handler on page 426
Extend the `CompositeControlAuthenticationHandler` class to combine the decisions from multiple control authentication handlers.

Configuring authentication handlers

Authentication handlers and the order that the Diffusion server calls them in are configured in the `Server.xml` configuration file.

To configure authentication handlers for your server, edit the `Server.xml` configuration file to include the following elements:

```
<security>
 <authentication-handlers>
  <authentication-handler class="com.example.LocalLDAPHandler" />
  <control-authentication-handler handler-name="RemoteHandler" />
 </authentication-handlers>
</security>
```

Ordering your configuration handlers

The order of handlCer elements within the `<authentication-handlers>` element defines the order in which the authentication handlers are called. In the preceding example, `localLDAPHandler` is called first. If `localLDAPHandler` returns an ABSTAIN result, `RemoteHandler` is called next.

Order your authentication handlers from least to most restrictive and configure your handlers to abstain unless they are to explicitly allow or deny the authentication request.

For more information, see Authentication on page 415.

COnfiguring local authentication handlers

Configure local authentication handlers by using the `<authentication-handler/>` element. The value of the attribute `class` is the class name for the handler.

Configuring control authentication handlers

Configure control authentication handlers are configured by using the `<control-authentication-handler/>` element. The value of the attribute `handler-name` is the name by which the handler was registered by the control client. Control clients use the AuthenticationControl feature to register the handler and passing the binding name as a parameter.

If no control client has registered a control authentication handler with the name defined in the configuration file, the response for that handler is ABSTAIN.

Multiple control clients can register a control authentication handler with the same name. Registering a control authentication handler from multiple clients gives the following advantages:

- If one of the control clients becomes unavailable, another can handle the authentication request.
- Control clients can be changed or updated without affecting the authentication behavior.
- Authentication requests can be load balanced between the control clients.

> **Note:** To register a control authentication handler, a control client must first connect to and authenticate with the server. We recommend that you configure a local authentication handler in the `Server.xml` file to authenticate the control client.

Related Links

Authentication API on page 428
The interfaces that are used for authentication are contained in the packages `com.pushtechnology.diffusion.client.security.authentication` and `com.pushtechnology.diffusion.client.types`.

AuthenticationControl on page 267
Use the AuthenticationControl feature to enable a control client session to authenticate other clients.

server on page 444
Server.xml - defines general Diffusion server properties as well as multiplexers, security, conflation, client queues, and thread pools.

Developing a local authentication handler

Implement the `AuthenticationHandler` interface to create a local authentication handler.

About this task

Local authentication handlers can be implemented only in Java.

> **Note:** Where *`c.p.d`* is used in package names, it indicates `com.pushtechnology.diffusion`.

Procedure

1. Create a Java class that implements `AuthenticationHandler`.

```
package com.example;

import com.pushtechnology.diffusion.client.details.SessionDetails;
import
 com.pushtechnology.diffusion.client.security.authentication.AuthenticationHandler
import com.pushtechnology.diffusion.client.types.Credentials;

public class ExampleAuthenticationHandler implements
 AuthenticationHandler{

    public void authenticate(String principal, Credentials credentials,
             SessionDetails sessionDetails, Callback callback) {

        // Logic to make the authentication decision.

        // Authentication decision
        callback.abstain();

        // callback.deny();
        // callback.allow();

    }
```

```
}
```

a) Ensure that you import `Credentials` from the *`c.p.d`*`.client.types` package, not the *`c.p.d`*`.api` package.
b) Implement the `authenticate` method.
c) Use the `allow`, `deny`, or `abstain` method on the `Callback` object to respond with the authentication decision.

2. Package your compiled Java class in a JAR file and put the JAR file in the `ext` directory of your Diffusion installation.

   This includes the authentication handler on the server classpath.

3. Edit the `etc/Server.xml` configuration file to point to your authentication handler.

   Include the `authentication-handler` element in the list of authentication handlers. The order of the list defines the order in which the authentication handlers are called. The value of the `class` attribute is the fully qualified name of your authentication handler class. For example:

```
<security>
    <authentication-handlers>

        <authentication-handler
 class="com.example.ExampleAuthenticationHandler" />

    </authentication-handlers>
</security>
```

4. Start or restart the Diffusion server.

   - On UNIX-based systems, run the `diffusion.sh` command in the *`diffusion_installation_dir`*`/bin` directory.
   - On Windows systems, run the `diffusion.bat` command in the *`diffusion_installation_dir`*`\bin` directory.

Related Links

Authentication handlers on page 418
You can implement authentication handlers that authenticate client connections to the Diffusion server or perform an action that requires authentication.

Authentication on page 415
You can implement and register handlers to authenticate clients when the clients try to perform operations that require authentication.

AuthenticationControl on page 267
Use the AuthenticationControl feature to enable a control client session to authenticate other clients.

Developing a composite authentication handler

Extend the `CompositeAuthenticationHandler` class to combine the decisions from multiple authentication handlers.

About this task

If there are several, discrete authentication steps that must always be performed in the same order, packaging them as a composite authentication handler simplifies the server configuration.

This example describes how to use a composite authentication handler to call multiple local authentication handlers in sequence.

Procedure

1. Create the individual authentication handlers that your composite authentication handler calls.

   You can follow step 1 on page 421 in the task Developing a local authentication handler on page 421.

In this example, the individual authentication handlers are referred to as `HandlerA`, `HandlerB`, and `HandlerC`.

2. Extend the `CompositeAuthenticationHandler` class.

```
package com.example;

import com.example.HandlerA;
import com.example.HandlerB;
import com.example.HandlerC;

import
 com.pushtechnology.diffusion.client.security.authentication.CompositeAuthenticati

public class CompositeHandler extends CompositeAuthenticationHandler {

    public CompositeHandler() {
        super(new HandlerA(), new HandlerB(), new HandlerC());
    }

}
```

   a) Import your individual authentication handlers.
   b) Create a no-argument constructor that calls the super class constructor with a list of your individual handlers.

3. Package your compiled Java class in a JAR file and put the JAR file in the `ext` directory of your Diffusion installation.

   This includes the composite authentication handler on the server classpath.

4. Edit the `etc/Server.xml` configuration file to point to your composite authentication handler.

   Include the `authentication-handler` element in the list of authentication handlers. The order of the list defines the order in which the authentication handlers are called. The value of the `class` attribute is the fully qualified class name of your composite authentication handler. For example:

```
<security>
  <authentication-handlers>

    <authentication-handler class="com.example.CompositeHandler" />

  </authentication-handlers>
</security>
```

5. Start the Diffusion server.

   - On UNIX-based systems, run the `diffusion.sh` command in the *`diffusion_installation_dir`*`/bin` directory.
   - On Windows systems, run the `diffusion.bat` command in the *`diffusion_installation_dir`*`\bin` directory.

Results

When the composite authentication handler is called, it calls the individual authentication handlers that are passed to it as parameters in the order they are passed in.

- If an individual handler responds with ALLOW or DENY, the composite handler responds with that decision to the server.
- If an individual handler responds with ABSTAIN, the composite handler calls the next individual handler in the list.
- If all individual handlers respond with ABSTAIN, the composite handler responds to the server with an ABSTAIN decision.

Related Links

Authentication handlers on page 418

You can implement authentication handlers that authenticate client connections to the Diffusion server or perform an action that requires authentication.

Authentication on page 415
You can implement and register handlers to authenticate clients when the clients try to perform operations that require authentication.

AuthenticationControl on page 267
Use the AuthenticationControl feature to enable a control client session to authenticate other clients.

Developing a control authentication handler

Implement the `ControlAuthenticationHandler` interface to create a control authentication handler.

About this task

A control authentication handler can be implemented in any language where the Diffusion Unified API includes the AuthenticationControl feature.

For more information, see AuthenticationControl on page 267.

This example demonstrates how to implement a control authentication handler in Java.

> **Note:** Where *`c.p.d`* is used in package names, it indicates `com.pushtechnology.diffusion`.

Procedure

1. Edit the `etc/Server.xml` configuration file to include a name that the control authentication handler can register with.

   Include the `control-authentication-handler` element in the list of authentication handlers. The order of the list defines the order in which the authentication handlers are called. The value of the `handler-name` attribute is the name that your control authentication handler registers as. For example:

   ```
   <security>
     <authentication-handlers>
       <-- Include a local authentication handler that can authenticate
    the control client -->
       <authentication-handler class="com.example.LocalHandler" />

       <-- Register your control authentication handler -->
       <control-authentication-handler handler-name="example-control-
   authentication-handler" />

     </authentication-handlers>
   </security>
   ```

   The control client that registers your control authentication handler must first authenticate with the server. Configure a local authentication handler that allows the control client to connect.
2. Start the Diffusion server.
   - On UNIX-based systems, run the `diffusion.sh` command in the *`diffusion_installation_dir`*`/bin` directory.
   - On Windows systems, run the `diffusion.bat` command in the *`diffusion_installation_dir`*`\bin` directory.
3. Create a Java class that implements `ControlAuthenticationHandler`.

   ```
   package com.example.client;

   import com.pushtechnology.diffusion.client.details.SessionDetails;
   ```

```
import
 com.pushtechnology.diffusion.client.features.control.clients.AuthenticationControl
import com.pushtechnology.diffusion.client.types.Credentials;

public class ExampleControlAuthenticationHandler implements
 ControlAuthenticationHandler{

    public void authenticate(String principal, Credentials credentials,
            SessionDetails sessionDetails, Callback callback) {

        // Logic to make the authentication decision.

        // Authentication decision
        callback.abstain();

        // callback.deny();
        // callback.allow();

    }

    @Override
    public void onActive(RegisteredHandler handler) {

    }

    @Override
    public void onClose() {

    }


}
```

a) Ensure that you import `Credentials` from the *`c.p.d`*`.client.types` package, not the *`c.p.d`*`.api` package.
b) Implement the `authenticate` method.
c) Use the `allow`, `deny`, or `abstain` method on the `Callback` object to respond with the authentication decision.
d) You can override the `onActive` and `onClose` to include actions the control authentication handler performs when the control client opens its connection to the server and when the control client closes its session with the servers.

   For example, when the control client session becomes active, the control authentication handler uses the `onActive` method to open a connection to a database. When the control client session is closed, the control authentication handler uses the `onClose` method to close the connection to the database.

4. Create a simple control client that registers your control authentication handler with the server.

```
package com.example.client;

import com.example.client.ExampleControlAuthenticationHandler;
import com.pushtechnology.diffusion.client.Diffusion;
import
 com.pushtechnology.diffusion.client.details.SessionDetails.DetailType;
import
 com.pushtechnology.diffusion.client.features.control.clients.AuthenticationControl
import com.pushtechnology.diffusion.client.session.Session;
import com.pushtechnology.diffusion.client.session.SessionFactory;

import java.util.EnumSet;

public class ExampleControlClient {

    public static void main(String[] args) {
```

```
        final Session session;

        // Create the control client session
        SessionFactory sf = Diffusion.sessions();
        session = sf.principal("ControlClient1")
                    .passwordCredentials("Passw0rd")
                    .open("dpt://localhost:8081");

        // Get the AuthenticationControl feature
        AuthenticationControl authControl =
 session.feature(AuthenticationControl.class);

        // Use the AuthenticationControl feature to register your
 control authentication
        // handler with the name that you configured in Server.xml
        authControl.setAuthenticationHandler("example-control-
authentication-handler",
                EnumSet.allOf(DetailType.class), new
 ExampleControlAuthenticationHandler());

        // Start the control client session
        session.start();

    }
}
```

a) Create a session.
b) Use the session to get the `AuthenticationControl` feature.
c) Use the `AuthenticationControl` feature to register your control authentication handler, `ExampleControlAuthenticationHandler`, using the name that you configured in the `etc/Server.xml` configuration file, “example-control-authentication-handler”.
d) Start the control client session.

For more information about developing a control client, see Control client on page 226

5. Start your client.

   It connects to the server and registers the control authentication handler with the name “example-control-authentication-handler”.

Related Links

Authentication handlers on page 418
You can implement authentication handlers that authenticate client connections to the Diffusion server or perform an action that requires authentication.

Authentication on page 415
You can implement and register handlers to authenticate clients when the clients try to perform operations that require authentication.

AuthenticationControl on page 267
Use the AuthenticationControl feature to enable a control client session to authenticate other clients.

Developing a composite control authentication handler

Extend the `CompositeControlAuthenticationHandler` class to combine the decisions from multiple control authentication handlers.

About this task

Using a composite control authentication handler reduces the number of messages that are sent between the server and the control client to perform authentication.

This example describes how to use a composite control authentication handler as part of a control client remote from the server.

Procedure

1. Edit the `etc/Server.xml` configuration file to point to your composite control authentication handler.

   Include the `control-authentication-handler` element in the list of authentication handlers. The order of the list defines the order in which the authentication handlers are called. The value of the `handler-name` attribute is the name that your composite control authentication handler registers as. For example:

   ```
   <security>
     <authentication-handlers>
       <-- Include a local authentication handler that can authenticate
    the control client -->
       <authentication-handler class="com.example.LocalHandler" />

       <-- Register your composite control authentication handler -->
       <control-authentication-handler handler-name="example-composite-
   control-authentication-handler" />

     </authentication-handlers>
   </security>
   ```

   The control client that registers your control authentication handler must first authenticate with the server. Configure a local authentication handler that allows the control client to connect.
2. Start the Diffusion server.
   - On UNIX-based systems, run the `diffusion.sh` command in the *`diffusion_installation_dir`*`/bin` directory.
   - On Windows systems, run the `diffusion.bat` command in the *`diffusion_installation_dir`*`\bin` directory.
3. Create the individual control authentication handlers that your composite control authentication handler calls.

   You can follow step 3 on page 424 in the task Developing a control authentication handler on page 424.

   In this example, the individual control authentication handlers are referred to as `HandlerOne`, `HandlerTwo`, and `HandlerThree`.
4. Extend the `CompositeControlAuthenticationHandler` class.

   ```
   package com.example.client;

   import com.example.client.HandlerOne;
   import com.example.client.HandlerTwo;
   import com.example.client.HandlerThree;

   import
    com.pushtechnology.diffusion.client.features.control.clients.CompositeControlAuth

   public class ExampleHandler extends
    CompositeControlAuthenticationHandler {

       public ExampleHandler() {
           super(new HandlerOne(), new HandlerTwo(), new HandlerThree());
       }

   }
   ```

   a) Import your individual control authentication handlers.
   b) Create a no-argument constructor that calls the super class constructor with a list of your individual handlers.

5. Create a simple control client that registers your composite control authentication handler with the server.

   You can follow step 4 on page 425 in the task Developing a control authentication handler on page 424.

   Ensure that you register your composite control authentication handler, `ExampleHandler`, using the name that you configured in the `etc/Server.xml` configuration file, "example-composite-control-authentication-handler".
6. Start your client.

   It connects to the server and registers the composite control authentication handler.

Results

When the control client session starts, the composite control authentication handler calls the `onActive` methods of the individual control authentication handlers in the order in which they are passed in to the composite handler.

When the composite control authentication handler is called, it calls the individual control authentication handlers that are passed to it as parameters in the order they are passed in.

- If an individual handler responds with ALLOW or DENY, the composite handler responds with that decision to the server.
- If an individual handler responds with ABSTAIN, the composite handler calls the next individual handler in the list.
- If all individual handlers respond with ABSTAIN, the composite handler responds to the server with an ABSTAIN decision.

When the control client session closes, the composite control authentication handler calls the `onClose` methods of the individual control authentication handlers in the order in which they are passed in to the composite handler.

Related Links

Authentication handlers on page 418
You can implement authentication handlers that authenticate client connections to the Diffusion server or perform an action that requires authentication.

Authentication on page 415
You can implement and register handlers to authenticate clients when the clients try to perform operations that require authentication.

AuthenticationControl on page 267
Use the AuthenticationControl feature to enable a control client session to authenticate other clients.

Authentication API

The interfaces that are used for authentication are contained in the packages `com.pushtechnology.diffusion.client.security.authentication` and `com.pushtechnology.diffusion.client.types`.

Local authentication handlers can be written only in Java and must be present on the server's classpath to be used. Control authentication handlers use the Unified API. You can write control authentication handlers in any language for which the client library supports the AuthenticationControl feature.

This section contains information only about those parts of the API that are used to write a local authentication handler. For more information about those parts of the API specific to control authentication handlers, see AuthenticationControl on page 267.

> **Note:** Where there notation `c.p.d` is used in class or package names, it indicates `com.pushtechnology.diffusion`.

API documentation for the authentication API is available at the following location: Java Unified API documentation

Classes and interfaces in the `com.pushtechnology.diffusion.client.security.authentication` package

The following table describes classes and interfaces that you must use when writing an authentication handler:

Table 82: Classes and interfaces in `c.p.d.client.security.authentication`

Interface	Description
`AuthenticationHandler`	Your authentication handler must implement this interface. **Note:** Your authentication handler can have either a no-argument constructor or a constructor that takes a single argument: `ServerConfig`. Use the single argument constructor when you want your authentication handler to access the server configuration, (for example, to discover the name of the server in which the handler is running).
`CompositeAuthenticationHandler`	Your composite authentication handler must extend this class.
`AuthenticationHandler.Callback`	Your authentication handler must respond to authentication requests by calling one of the methods on this interface: `allow()`, `deny()`, or `abstain()`.

Interfaces in the `c.p.d.client.types` package

The following table describes interfaces that you must use when writing an authentication handler:

Table 83: Interfaces in `c.p.d.client.types`

Interface	Description
`Credentials`	Your authentication handler receives the principal's credential information as a ByteArray wrapped in an object that implements the `Credentials` interface. You can use the `toBytes()` method to retrieve the byte information. The authentication handler must be able to interpret the bytes in the array appropriately. Credentials can have one of the following types: • NONE – an empty byte array • PLAIN_PASSWORD – text encoded in UTF-8 format • CUSTOM – any other application-specific credentials information The type is returned by the `getType()` method.

Related Links

Unified API on page 240
The Diffusion Unified application programming interface (API) provides a consistent interface to the Diffusion server, whichever role your client application performs: standard client or control client.

Authentication handlers on page 418

You can implement authentication handlers that authenticate client connections to the Diffusion server or perform an action that requires authentication.

Authorization handlers

An authorization handler can control authorization and permissioning for clients and users.

An authorization handler is a user-written Java class that must implement the `AuthorisationHandler` interface in the Classic API.

Such a handler can be used to restrict access of clients according to any criteria that is appropriate. One capability within Diffusion is for a client to be able to specify Credentials when they connect that can be checked by the authorization handler.

The handler can either be specified in `etc/Server.xml` in which case it is loaded when the server starts or can be set programmatically within a publisher using the `Publishers.setAuthorisationHandler` method.

There can only be one handler and it is system wide across all publishers, although you can have authorization at the publisher level.

If an authorization handler is not specified, credentials sent by a client are assumed to be valid. A publisher has access to the credentials to perform finer-grained authorization, if required.

The authorization handler interface has the following methods:

Table 84: Authorization handler methods

DEPRECATED: `canConnect(Client)`	This method is called to establish whether the client can connect and is called before any client validation policy is called.
`canSubscribe(Client, Topic)`	This method is called when a client subscribes to a topic. If topic information is sent with the connection, this method is called after the `canConnect` method. **Note:** This is called for every topic being subscribed to, even if subscribed as a result of a topic selector being specified. However (by default), if a topic is rejected by this method, it is not called again for any children (or descendants) of the topic.
`canSubscribe(Client, TopicSelector)`	This method is called when a client attempts to subscribe to a topic selector pattern (as opposed to a simple topic name). If topic information is sent with the connection, this method is called after the `canConnect` method.
`canFetch(Client, Topic)`	This method is called when a client sends a fetch request to obtain the current state of a topic. **Note:** This is called for every topic being fetched, even if fetched as a result of a topic selector being specified. However (by default), if a topic is rejected by this method, it is not be called again for any children (or descendants) of the topic.

`canFetch(Client, TopicSelector)`	This method is called when a client attempts to fetch topics using a topic selector pattern (as opposed to a simple topic name).
`canWrite(Client, Topic)`	This method is called when a client sends a message on a given topic, if false is returned the message is ignored, and the publisher will not be notified of the message. When implementing this method, be aware that performance can be impacted if many clients send messages or if a few clients send large messages.
DEPRECATED: `credentialsSupplied(Client, Credentials)`	This method is called when a client submits credentials after connection. It can be used to validate the credentials and must return true if the credentials are OK. If this returns false, a Credentials Rejected message are sent back to the client.

Authentication

> **Note:** The use of authorization handlers for authentication is deprecated. We recommend that you re-implement your authentication logic using authentication handlers. For more information, see Authentication handlers on page 418.

When a client connects to Diffusion it has the option of supplying user credentials. These credentials are basically tokens called username and password. These tokens can be used for any purpose. When `canConnect` is called, you can get the credentials from the `Client` object.

An example of this is:

```
   public boolean canConnect(Client client) {
Credentials creds = client.getCredentials();

// No creds supplied, so reject the connection
if (creds == null) {
    return false;
}

String username = creds.getUsername().toLowerCase();
```

If the credentials are null, none were supplied, which is different from empty credentials.

Clients can connect without credentials and submit them later or replace the credentials at any time whilst connected. The authorization handler is notified when new credentials are submitted and can choose to set the new credentials on the client.

The `Credentials` class has username and password attributes, but also allows for an attachment. It is here that a user normally sets any security object required. Returning true will allow the user to connect, returning false will result in the client connection being refused.

Subscription authorization

SUbscription authorization is the allowing of a client to subscribe to a topic. In this case the `canSubscribe` is called. Returning true here allows the publisher to have any topic loaders and subscription methods called. Returning false will not notify the client that the subscription was invalid.

```
public boolean canSubscribe(Client client, Topic topic) {

    // Everyone has access to the top level topic
    if (topic.getName().equals(CHAT_TOPIC)) {
```

```
            return true;
        }

        User user = (User) client.getCredentials().attachment();

        return user.isAllowedRoom(topic.getNodeName());
    }
```

Authorization handler

Authorization at the publisher level can also be achieved. This is required if there are many publishers running within the same Diffusion Server and they have different security settings. The following code example works if the publishers all implement `AuthorisationHandler`

```
public boolean canSubscribe(Client client, Topic topic) {

    AuthorisationHandler handler =
            (AuthorisationHandler)Publishers.getPublisherForTopic(topic);

    // Call the publisher in question
    return handler.canSubscribe(client, topic);
}
```

Permissioning

The permissioning process governs whether a client is able to send messages to a publisher, or in other words, is the topic read only. This is handled by the `canWrite` method. Again a good pattern might be to look at the credentials attachment object to see if this is permissible.

```
public boolean canWrite(Client client, Topic topic) {
 User user = (User) client.getClientCredentials().attachment();
 return user.canWriteMessages(topic);
}
```

Network security

This section describes how to deploy network security, which can be used in conjunction with data security.

Secure clients

Below is a table of the clients and whether they can connect securely through SSL and/or HTTPS.

Table 85: Client security

Client	DPTS	HTTPS
.NET	✓	✓
Java	✓	✓
Flash	✓	✓
Silverlight	✗	✓
WebSocket	✓	✗

Client	DPTS	HTTPS
XHR	✗	✓
Android	✓	✗
iOS	✓	✗

You can have Diffusion server master/slave connections over DPTS as well.

Web server configuration

The web server can be configured in your test environment to allow you to deploy and undeploy DAR files by using a web service. By default this capability is not enabled.

For security, if you choose to enable this web service in your production environment, you must restrict access to the *diffusion-url*`/deploy` URL by other means. For example, by setting up restrictions in your firewall.

To configure the web server, use the `WebServer.xml` file. For more information, see web-servers on page 465. An example of this file is provided in the `/etc` directory of the Diffusion installation. The XSD is provided in the `/xsd` directory of the Diffusion installation.

Connector configuration

If secure connections are required, Diffusion connectors must be configured to support DPTS and/or HTTPS. Any connector can accept DPTS connections. A connector does not have to be dedicated to only DPTS connections. DPTS connections can also accept clear-text connections. To enable DPTS connections a keystore entry is required in the connector configuration. This informs the connector that it is enabled for DPTS type connections. If HTTPS is required, a keystore section and a web-server entry are also required, even for secure Diffusion clients.

To configure the connectors, use the `Connectors.xml` file. For more information, see connectors on page 456. An example of this file is provided in the `/etc` directory of the Diffusion installation. The XSD is provided in the `/xsd` directory of the Diffusion installation.

Keystores

When connecting, an incorrect keystore results in SSL errors. One of these being no cipher suites in common.

The following steps use the Java Keytool to create a keystore. The steps can vary depending on your certificate authority. For more information, refer to your certificate authority's documentation.

1. Generate a key and place it in your keystore.

```
keytool -genkeypair -alias my_alias -keyalg RSA -keystore  -keysize
```

2. Generate a certificate signing request (CSR) file.

```
keytool -certreq -keyalg RSA -alias my_alias -file certreq.csr -
keystore
```

3. Send the CSR file to your certificate authority.
4. Receive the signed certificate from your certificate authority.
5. Install any intermediate certificates that you require.

```
keytool -import -trustcacerts -alias intermediate_alias -keystore  -
file intermediate_certificate_file.crt
```

6. Install your own certificate. Use the same alias as when you generated the key and the signing request.

```
keytool -import -trustcacerts -alias my_alias -keystore  -
file certificate_file.crt
```

Related Links

http://www.networking4all.com/en/support/ssl+certificates/manuals/java/java+based+webserver/keytool+commands/

Chapter 15

Distribution

In this section:

- Publisher clients
- Distributed topics
- Distribution examples

Diffusion provides an infinitely scalable distributed solution which can involve any number of Diffusion servers and any number of publishers.

The following sections discuss distribution in more detail:

Publisher clients

>A publisher can connect upstream to a Diffusion server as if it were a client connecting to that server. The publisher can interact with the server like any other client. When a publisher acts in this way it is said to be acting as a publisher client.

Publisher client capabilities

From within the publisher a publisher client connection to a server is represented by the `PublisherServerConnection` interface. This is a subtype of `ServerConnection` which is the same interface used by connections to servers from external clients. A publisher acting as a client has all the same capabilities as any external client. For example, it can subscribe to Topics on that server or can send messages to topics on that server.

For full details of the capabilities see the issued API documentation for the ServerConnection interface.

Publisher client connection

A publisher connects to a server listening on a given port at a specified host. The host and port correspond to a client connector configured at that server.

Connecting using API

A publisher can register a connection to a server using `addServerConnection` and then use the `ServerConnection.connect` method to make a connection. Within a publisher the following code creates a new server connection to the localhost on port 8080:

```
PublisherServerConnection conn =
 this.addServerConnection("ServerLocalName","localhost",8080);
```

This connection is named ServerLocalName, this is the name that the publisher uses to refer to the connection. The `addServerConnection` method can take a `ServerDetails` object instead of the host and port.

```
PublisherServerConnection conn =
 this.addServerConnection("ServerLocalName",ConnectionFactory.createServerDetails("dp
localhost:8080"));
```

See the API documentation for more information about how to use `ConnectionFactory.createServerDetails`.

A failure policy can be set for the connection before connecting which defines the action to take if the connection fails or is lost at some point after connection.

The connection is not opened as soon as it is declared, it must be opened explicitly with `connect`. Once the connection has been opened, the connection can be used to subscribe to topics.

When connected the publisher is notified through the `serverConnected` method and the publisher is able to subscribe to topics on the server as required.

A complete example of how to connect using the API.

```
PublisherServerConnection conn =
 this.addServerConnection("ServerLocalName","localhost",8080);
conn.setFailPolicy(PublisherServerConnectionFailPolicy.RETRY);
try {
 String clientId = conn.connect();
}
catch (APIException e) {
 ...
}
```

Configured connection

It might be more convenient to configure the servers that a publisher automatically connects to in the `etc/Publishers.xml` file. In this case any servers that are declared and connected before `initialLoad` is called, and all connections that have not been explicitly connected to are automatically connected after `initialLoad`.

A failure policy can be configured so that if a connection fails it either stops the publisher or attempt to periodically retry the connection. The default policy is to ignore a failed connection and allow the publisher to deal with it.

Any number of server connections can be configured as elements inside the publisher element of the `etc/Publishers.xml`. The following is a minimal example, all elements of server in it are required. Refer to the XSD for further information.

```
<publisher name="ChildPublisher">
 ...
 <server name="ServerLocalName">
  <host>localhost</host>
  <port>8081</port>
  <input-buffer-size>4k</input-buffer-size>
  <output-buffer-size>64k</output-buffer-size>
  <fail-policy>RETRY</fail-policy>
 </server>
</publisher>
```

Connection failure policy

A connection failure policy can be defined for a server connection to indicate what happens if the connection is lost at some point after the connection is made. For configured connections the failure policy also applies to failure to connect during startup.

A failure policy can be set at any time on a connection using the `setFailPolicy` method.

For configured connections the failure policy can be specified in `etc/Publishers.xml`

The following policy options are available:

DEFAULT

For a configured connection, failure to connect at startup is logged and the publisher can determine the state of such connections after `initialLoad` (for example, in `publisherStarted`) using the `ServerConnection.getState` method.

If a connection is lost, the publisher is notified through the `serverDisconnected` method but no further action is taken.

CLOSE

For a configured connection, failure to connect at startup causes the publisher to be stopped.

For any type of connection, if the publisher loses its connection to the server, the publisher is closed.

RETRY

For a configured connection that fails during startup or any connection that is lost after connection is made, the connection is periodically retried until it is reconnected.

The default retry interval is 5 seconds but this can be specified if required.

Using the connection

Managing a publisher's server connections is done using the name of the connection to identify it.

Any named connection can be retrieved, regardless of whether it was created using the API or the configuration.

```
PublisherServerConnection conn =
 this.getServerConnection("ServerLocalName");
```

Any named connection can be closed and removed from the publisher using `removeServerConnection`.

The publisher can test to see if a name has been used to describe a server connection using `hasServerConnection`.

When a publisher connects to a server the `serverConnected` method is called. This uses the ServerConnection interface to pass the connection information. This method can be overridden so that you can implement custom behavior when a server is connected. Once connected to the server the publisher can interact with its topics, it can subscribe and send messages.

```
@Override
protected void serverConnected(ServerConnection conn) {
 if (conn.getName().equals("ServerLocalName")) {
  conn.subscribe("TopicName");
  ...
 }
}
```

The `serverDisconnected` method is called to notify the publisher a server has disconnected. This uses the ServerConnection interface to pass the connection information. This method can be overridden so that you can implement custom behavior when a server is connected.

```
@Override
protected void serverDisconnected(ServerConnection conn) {
 if (conn.getName().equals("ServerLocalName")) {
  ...
 }
}
```

The publisher is responsible for processing messages that are received from the server using the method. This uses the ServerConnection interface to pass the connection information.

```
@Override
protected void messageFromServer(ServerConnection conn, TopicMessage
 message) {
...
}
```

The ServerConnection can be used to construct and send messages to the server. It provides `createLoadMessage`, `createDeltaMessage` and `send` to communicate with the server. The server treats a publisher client as any other client and no special handling is required on its side.

Notifications

A publisher is notified of a connection to a server using its `serverConnected` method. This is the ideal place to subscribe to topics on the server.

Once connected a publisher receives messages from servers on its `messageFromServer` method which indicates from which server the message came.

A publisher receives notification of the change of status (for example, removal) of any topics it subscribed to on a server using the `serverTopicStatusChanged` method.

The publisher is notified of a lost connection on its `serverDisconnected` method.

Buffer sizes

For a configured connection the sizes of the input and output buffers to use for the socket connection can be specified in `etc/Publishers.xml`

The buffer sizes can also be specified (or changed) for a `ServerConnection` by obtaining the `ServerDetails` (using `getServerDetails`) and using `setInputBufferSize` and setOutputBufferSize. Call these methods before calling connect.

It is important that the buffer sizes specified are compatible with those at the other end of the connection. Remember that the input buffer matches the server's output buffer and the output buffer matches the server's input buffer.

Distributed topics

You can serve topics of the same name from more than one Diffusion server in your distributed environment.

Within Diffusion, Topics on page 126 are unique within any one server but there is no restriction on serving more than one server in a distributed environment having topics of the same name. Indeed, this is often required.

Diffusion itself does not impose any synchronization of topics across servers. This is to allow maximum flexibility of the configuration of topics. There is no reason why a publisher that subscribes (as a client) to the topic or topics of a publisher at another server has to duplicate those topics in its own server. However, in some circumstances (for example, distributors) this might be exactly what is required.

The behavior of distributed publishers is different to Event publishers that use the Classic API on page 224, where messages for a topic are routed to the publisher that owns the topic.

For example, a publisher can subscribe to a topic called Feeder on another server but that does not mean that there has to be a topic called Feeder on its own. The publisher can choose to transform the data received on the Feeder topic and publish it on an entirely different topic.

Topic replication

Where topic replication is required from the server that is connected to this can be greatly simplified by using `TopicNotifyTopicData` in the master server and the slave server can subscribe to the notifying topic and receive notifications of topics added. For more information, see Topic notify topic data on page 176

It is not necessary to replicate all topics at the master server as the notification facility can be used to notify some topic creations.

In the case where replication is required, full notifications must be requested. The `TopicDefinition` objects passed in the notifications can be used directly to create duplicate topics local to the publisher client.

Distribution examples

There are a number of distribution scenarios within Diffusion.

Distributors

This is the term used to represent the situation where there is a single publisher publishing messages to one or more other publishers in other Diffusion Servers.

In this case the one that distributes the messages is known as the distributor publisher and the others are recipients. The recipient publishers connect to the server of the distributor publisher and subscribe to its topics as publisher clients. Messages that the distributor publishes are then published to all clients which in this case are publishers.

This scenario can be used to balance client connections across a number of Diffusion servers all serving the same data.

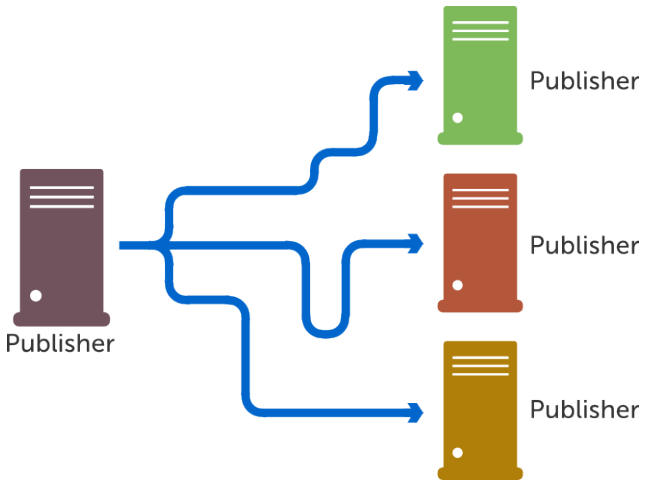


Figure 54: Distributors

Aggregators

Data aggregation

This is the term used to describe the situation where a single publisher aggregates messages received from one or more other publishers.

In this case the aggregator connects to more than one other Diffusion server and subscribes to topics on them.

This scenario can be used to aggregate data being received from more than one source providing a single point of contact for clients which see only the aggregated data.

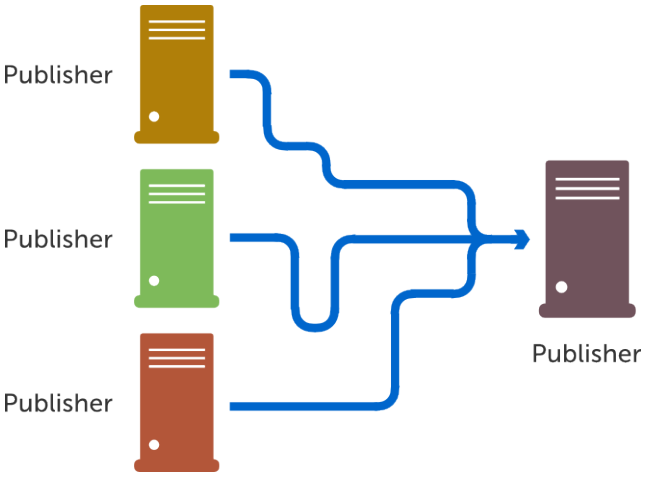


Figure 55: Aggregators

Mixed mode

The client/server relationship between publishers is not restricted to the above examples. You can set up a peer-to-peer network of publishers, which communicate with each other in any way required.

A publisher publishes to clients. Clients connected to a Diffusion server can be a mixture of publisher clients and other clients.

Chapter 16

Configuration

In this section:

- XML configuration
- Programmatic configuration

You can configure the Diffusion server using XML files which normally reside in the `etc` folder.

Alternatively, a Diffusion server can be instantiated in a Java application and configured programmatically. Some properties can also be changed at runtime programmatically from within publishers.

In a Java client environment certain properties can also be configured programmatically.

All properties (whether configured from XML or programmatically) are available to read programmatically from within the Java API.

XML configuration

Configuring a Diffusion server using XML property files

XML Property files

A Diffusion server is configured using a set of XML property files typically loaded from the `etc` folder. In a new Diffusion installation example versions of these files are provided which can be edited as required.

XML is used rather than standard property files due to the hierarchic nature and the ability to support repeating groups.

The Introspector has a built in configuration editor, which is able to load and save the configuration files remotely if required.

XSD files are issued that define the content of the XML property files and this section summarizes the XSD content.

Configuration path loading

You can pass a parameter to Diffusion upon startup so that files are not automatically loaded from the `etc` folder but loaded from a different folder. This folder does not have to contain the complete set of XML files, but the file is loaded from the specified folder first, if it exists. If it does not, Diffusion loads the configuration file from the `etc` folder. When Diffusion starts, it logs where each configuration file has been loaded from.

Classpath loading

When Diffusion starts, the `data` and `etc` folder are on the class path. The ext folder, and its sub-directories are scanned for jar files and class loaded. This means that you can add new jars to the Diffusion runtime, without having to edit the startup scripts. Caution must be taken when creating backup jars in the ext folder. Anything that is located in `.jar` is class loaded.

XML Value types

When XML values are loaded, the schema is checked so that we know that it is valid, but to aid configuration, there are some extra data types. When values are loaded, they are trimmed of leading and trailing white space.

Table 86: XML Value types

Data type	Meaning
push:boolean	true or false
push:string	String value
push:int	A number between -2,147,483,648 and 2,147,483,647
push:long	A number between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807
push:double	A number between 2-1074and (2-2-52)¬†.21023
push:port	A positive number but less than 65535
push:millis	A string that represents the number of milliseconds. Append the mnemonic for the time unit. The mnemonic can be either upper or lower case. **s** Seconds

Data type	Meaning
	m Minutes **h** Hours **d** Days 360000, 360s, 6m all represent 6 minutes
push:bytes	A string that represents the number of bytes. Append the mnemonic size unit. The mnemonic can be either upper or lower case. **k** Kilobytes **m** Megabytes **g** Gigabytes 6291456, 6144k, 6m, all represent 6 Megabytes
push:log-level	A log level can be FINEST, FINE, INFO, ADVICE, WARNING, or SEVERE
push:percent	A value that represents a percentage, this can have the trailing percent sign (%)
push:positiveNonZeroInt	A number between 1 and 2,147,483,647
push:positiveInt	A number between 0 and 2,147,483,647
push:positiveNonZeroLong	A number between 1 and 9,223,372,036,854,775,807
push:positiveLong	A number between 0 and 9,223,372,036,854,775,807
push:threadPriority	A number between 1 and 10
<element>	This notation is used to indicate a complex element type. It can also be List<element> to indicate a repeating property group.

Environmental values

When defining custom configurations, you can define environmental variables that can be reused in all XML property files. These variables can be defined in the `etc/Env.xml` property file to be used in all other property files. Suppose, for example, the `etc/Env.xml` file defines a `server-name` variable, with value `d-unit` as follows:

```
<env>
 <property name="server-name">d-unit</property>
</env>
```

The `server-name` variable can be used in all other property files, where the value `d-unit` is appropriate, either as a value for an attribute, as in

```
<server name={server-name}>,Ä¶</server>
```

or as a name for an element as in:

```
<server>{server-name}</server>
```

As a side remark, it is worth noting that names can be combined to provide malleable environmental variables. Suppose for instance `Env.xml` contains the following entries:

```
<env>
 <property name="server-name">myServer</property>
 <property name ="server-version">V2.0</property>
</env>
```

Then `server-name` and `server-version` can be combined, for instance within the same `etc/Env.xml`, as

```
<property name="server-and-version">{server-name}-{server-version}</
property>
```

and used in all other configuration files.

Obfuscated values

Obfuscation is a technique through which sensitive entries can be hidden in clear text. Within the Diffusion context, obfuscation can be used to hide password, and any other data deemed to be sensitive, to be included in configuration files.

To create obfuscated values you can use the **Property Obfuscator** dialog in the Diffusion plugin for Eclipse. For more information, see Property obfuscator on page 571. Obfuscated entries are identified by a `OB:` prefix in clear text.

Property files

The remainder of this topic defines the properties available in the major property files.

server

Server.xml - defines general Diffusion server properties as well as multiplexers, security, conflation, client queues, and thread pools.

server

All server properties

The following table lists the elements that an element of type `server` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
server-name	push:string	The server name is used to identify this server if running in a cluster. This is also used as a prefix for client IDs. If not specified, the local hostname is used.	0	1
max-message-size	push:bytes	The maximum message size in bytes. This defines the maximum message size (including headers) that can be handled.	1	1
default-load-message-capacity	push:bytes	The default capacity of a load message if not explicitly specified. If not supplied, a default of 4096 is used.	0	1
default-delta-message-capacity	push:bytes	The default capacity of a delta message if not explicitly specified. If not specified, a default of 1024 is used.	0	1

Name	Type	Description	Minimu occurre	Maximu occuren
message-length-size	push:int	Specifies the number of bytes utilized within message headers to accommodate the message length. This can have a value of 4, 2, or 1. This is a system-wide setting which must match for all components of a distributed Diffusion system. Client APIs automatically adapt to the size at the server they connect to. The size chosen depends upon the maximum message size that needs to be catered for. A value of 1 caters for messages up to 127 bytes in length (including headers), 2 caters for messages up to 32,767 bytes in length, and 4 can notionally cater for messages up to 2,147,483,647 bytes on length. If a value is not supplied, a default of 4 is used.	0	1
charset	push:string	The default character set to use for Diffusion message character conversions. See Java Encodings for the full list. If a value is not specified, a default of "UTF-8" is used.	0	1
multiplexers	multiplexers	Properties that define multiplexers and how they are used.	1	1
write-selectors	write-selectors	Properties that define write selectors and how they are used.	0	1
security	security	Properties relating to security (optional).	0	1
conflation	conflation	Conflation policies and topic to policy mappings.	0	1
client-queues	client-queues	Definitions of client queues.	1	1
connection-timeouts	connection-timeouts	Timeout values relating to connections. If a value is not specified, defaults are used.	0	1
date-formats	date-formats	Date and time formats. If a value is not specified, default formats are used.	0	1
thread-pools	thread-pools	Definitions of thread pools	1	1
whois	whois	Definition of the WhoIs lookup service. If a value is not specified, no WhoIs service runs.	0	1
auto-deployment	auto-deployment	Automatic deployment properties (optional).	0	1
geo-ip	geo-ip	Properties relating to the Geo IP lookup facility. If a value is not specified, defaults are used.	0	1
usr-lib	usr-lib	User libraries (optional).	0	1
hooks	hooks	User hooks used in the server (optional)	0	1

multiplexers

Multiplexer definitions.

The following table lists the elements that an element of type `multiplexers` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
client	push:string	Name of the client multiplexer definition. If this is not specified, the first multiplexer defined is used.	0	1
multiplexer-definition	multiplexer-definition	Multiplexer definition.	1	unbounded

multiplexer-definition

Multiplexer definition.

The following table lists the attributes that an element of type `multiplexer-definition` can have:

Name	Type	Description	Required
name	push:string	The multiplexer name.	true

The following table lists the elements that an element of type `multiplexer-definition` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
size	push:positiveInt	This is the number of multiplexer instances that start in readiness for clients to be assigned to. If there are going to be a large number of users, increase this number. If a value is not specified, a default of 2 is used.	0	1
thread-priority	push:threadPriority	This is the thread priority that the multiplexer threads run at. If a value is not specified, a default of 8 is used.	0	1
load-balancer	push:string	This is the load balancer to use for for assigning connecting clients to multiplexer instances. There are currently two implemented load balancers, 'RoundRobin' and 'LeastClients'. RoundRobin is fast, but cannot guarantee fairness of the connections per instance due to the randomness of disconnections. LeastClients guarantees fairness across the instances. If a value is not specified, a default of 'RoundRobin' is used.	0	1
latency-warning	push:millis	Multiplexers are critical to the operation of Diffusion. If there are too many clients assigned to too few multiplexer instances, there is a possibility of message latency. This is an optional flag which can be set to issue a warning if the multiplexer instance is taking too long in its operational cycle (see ServerNotificationListener in the publisher API). If this value is 0, this feature is not enabled. If this value is not supplied, a default of 0 is used.	0	1

Name	Type	Description	Minimu occurre	Maximu occuren
max-event-queue-size	push:positiveInt	This specifies the maximum size of the multiplexer event queue. This is the queue on which events from publishers are queued for multiplexers and the default value is normally more than adequate. If this queue fills, it can cause the publisher threads to block until they can enqueue events and in this case it might be necessary to increase the value. Typically, leave this value at the default of 128k.	0	1

write-selectors

Configuration for pool of write selectors. Write selectors are used for writing partially written buffers to slow-consuming clients. The default configuration is normally sufficient unless there are many slow-consuming clients - for example, SSL clients.

The following table lists the elements that an element of type `write-selectors` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
thread-priority	push:threadPriority	The priority to run selector threads with. Only change this value if advised by Push Technology.	0	1
size	push:positiveNonZeroInt	Number of selector threads to run. If this value is not specified, a default of 1 is used.	0	1
timeout	push:millis	Timeout for retry events in milliseconds. If this value is not specified, a default of 4000ms is used.	0	1
load-balancer	push:string	This is the load balancer to use for for assigning connecting clients to selection instances. There are currently two implemented load balancers, 'RoundRobin' and 'LeastClients'. RoundRobin is fast, but cannot guarantee fairness of the connections per instance because of the randomness of disconnections. 'LeastClients' guarantees fairness across the instances. If this value is not specified, a default of 'RoundRobin' is used.	0	1
queue-size	push:positiveNonZeroInt	Each selector thread is event driven. Events are first placed in a queue before being processed by the selector. The queue size is configurable and defaults to 1024.	0	1

hooks

User hooks used in the server.

The following table lists the elements that an element of type `hooks` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
startup-hook	push:string	This is the class name of a class that implements the interface com.pushtechnology.diffusion.api.publisher.ServerStartupHook. If specified, the hook is instantiated and the serverStarting method called when the server is starting, before the loading of publishers.	0	1
shutdown-hook	push:string	This is the class name of a class that implements the interface com.pushtechnology.diffusion.api.publisher.ServerShutdownHook. If specified, the hook is instantiated and the serverStopping method called when the server is stopping.	0	1

security

Server security properties.

The following table lists the elements that an element of type `security` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
authorisation-handler-class	push:string	This is the full name of a class, on the classpath, that implements the AuthorisationHandler interface in the Java publisher API. If specified, the handler is instantiated when the server starts and is called to authorize client connections, subscriptions, and fetch requests.	0	1
authentication-handlers	authentication-handlers		0	1

authentication-handlers

Authentication handlers, in order of decreasing precedence. The authentication handlers are called to authenticate new connections and changes to the principal associated with a session. Authentication handlers are configured in precedence order. Authentication succeeds if a handler returns "allow" and all higher precedence handlers (earlier in the order) return "abstain". Authentication fails if a handler returns "deny" and all higher precedence handlers return "abstain". If all authentication handlers return "abstain", the request is denied. After the outcome is known, the server might choose not to call the remaining handlers.

The following table lists the elements that an element of type `authentication-handlers` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
authentication-handler	server-authentication-handler		1	1
control-authentication-handler	control-authentication-handler		1	1

server-authentication-handler

An authentication handler hosted by the server. The handler is instantiated when the server starts. The class attribute specifies the fully qualified name of a handler implementation class that implements the com.pushtechnology.diffusion.client.security.authentication.AuthenticationHandler interface. The class must be available on the classpath.

The following table lists the attributes that an element of type `server-authentication-handler` can have:

Name	Type	Description	Required
class	push:string		true

control-authentication-handler

Control client sessions register control authentication handlers using an identifying name. A <control-authentication-handler> must be configured with a matching handler-name. Configure at most one <control-authentication-handler> for a handler-name.

The following table lists the attributes that an element of type `control-authentication-handler` can have:

Name	Type	Description	Required
handler-name	push:string		true

conflation

Conflation policies and topic to policy mappings.

The following table lists the elements that an element of type `conflation` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
default-conflation-policy	push:string	The default conflation policy. This specifies a conflation policy that is used for any topics that do not have explicit conflation policy mappings defined. If this is not specified, conflation does not occur for topics that do not have a policy mapping defined. If this value is specified, it must be the name of a defined policy.	0	1
conflation-policy	conflation-policy	Conflation policy.	0	unbounded
topic-conflation	topic-conflation	A mapping between a topic (or topic selector pattern) and a policy.	0	unbounded

conflation-policy

Conflation policy.

The following table lists the attributes that an element of type `conflation-policy` can have:

Name	Type	Description	Required
name	push:string	The conflation policy name.	true

The following table lists the elements that an element of type `conflation-policy` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
mode	push:string	The conflation mode. This can have the value 'replace' or 'append'. If a value is not specified, a default of 'replace' is assumed. The value 'replace' means that when a matching message is found, the message is replaced by the new (or merged) message in its current queue position. The value 'append' means that when a matching message is found, the message is removed from its current queue position and the new (or merged) message is appended to the end of the queue. This option preserves message ordering but there is the danger that messages are constantly sent to the end of the queue.	0	1
matcher	push:string	The full class name of a message matcher of type com.pushtechnology.diffusion.api.conflation.MessageMatcher. If a class is not supplied, a default matcher that matches by topic name is used.	0	1
merger	push:string	The full class name of a message merger of type com.pushtechnology.diffusion.api.conflation.MessageMerger. If a class is not supplied, no merging with the new message occurs. The new message either replaces the existing one or the existing one is removed and the new one appended to the end of the queue depending upon the mode.	0	1

topic-conflation

A mapping between a topic (or topic selector pattern) and conflation policy.

The following table lists the elements that an element of type `topic-conflation` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
topic	push:string	The name of a topic or a topic selector pattern that indicates the topic or topics that the specified conflation policy is applied to.	1	1
policy	push:string	The name of a configured conflation policy that is applied to the specified topic or topics.	1	1

client-queues

Client queue definitions.

The following table lists the elements that an element of type `client-queues` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
default-queue-definition	push:string	The name of the queue definition to use by default. Connectors that do not explicitly specify a queue definition use the one specified here.	1	1
queue-definition	queue-definition	Queue definition.	1	unbounded

queue-definition

This defines the properties of a client queue.

The following table lists the attributes that an element of type `queue-definition` can have:

Name	Type	Description	Required
name	push:string	The queue definition name.	true

The following table lists the elements that an element of type `queue-definition` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
max-depth	push:positiveInt	The maximum depth of the queue. If the number of messages queued for a client exceeds this number, the server disconnects the client.	1	1
conflates	push:boolean	Specifies whether conflation is applied to all clients using this queue definition. If this value is not specified, conflation is not applied by default.	0	1
upper-threshold	push:percent	This specifies a percentage of the maximum queue size and if this value is reached then any listeners (see ClientListener in the publisher API) are notified. Notification occurs only once and does not occur again until the queue has returned to the lower threshold. If this value is not specified, no upper limit notification occurs.	0	1
lower-threshold	push:percent	This specifies a percentage of the maximum queue size and indicates the level at which listeners (see ClientListener in the publisher API) are notified after an upper limit notification has occurred and the queue size has dropped back to the specified lower limit. If this value is not specified, no lower limit notification occurs.	0	1
auto-fragment	push:boolean	If a message is too large to fit into the output buffer and this option is set to true, the message is automatically fragmented to 80% of the output buffer size. As this happens on a per-client basis, it might be highly inefficient. Consider creating your messages inside your publisher with	0	1

Name	Type	Description	Minimu occurre	Maximu occuren
		fragmentation options instead. If this value is not specified, a default of false is assumed.		

connection-timeouts

Connection-related timeouts.

The following table lists the elements that an element of type `connection-timeouts` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
write-timeout	push:millis	This is the time in milliseconds that is allotted to performing a single write to a client. If the write has not completed within this limit, the client is disconnected. If this value is not specified, a default of 2s is used.	0	1
connection-timeout	push:millis	This is the time in milliseconds allowed for a connection to take place and complete its handshake processing. If this value is not specified, a default of 2s is used.	0	1

date-formats

Date and time formats.

The following table lists the elements that an element of type `date-formats` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
date	push:string	The format used when displaying dates. Specify the format according to the Java SimpleDateFormat specification. If a format is not specified, a default of "yyyy-MM-dd" is used.	0	1
time	push:string	The format used when displaying times. Specify the format according to the Java SimpleDateFormat specification. If a format is not specified, a default of "HH:mm:ss" is used.	0	1
date-time	push:string	The format used when displaying date and time. Specify the format according to the Java SimpleDateFormat specification. If a format is not specified, a default of "yyyy-MM-dd HH:mm:ss" is used.	0	1
timestamp	push:string	The format used when displaying a timestamp - for example, in a log - to millisecond precision. Specify the format according to the Java SimpleDateFormat specification. If a format is not specified, a default of "yyyy-MM-dd HH:mm:ss.SSS" is used.	0	1

thread-pools

Thread pools.

The following table lists the elements that an element of type `thread-pools` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
inbound	push:string	Name of the inbound thread pool definition.	1	1
outbound	push:string	This property is deprecated. Do not use.	0	1
background-thread-size	push:int	Number of threads to use for the background thread pool. If a value is not specified, a default of 10 is used.	0	1
thread-pool-definition	thread-pool-definition	Thread pool definition.	1	unbounded

thread-pool-definition

Thread pool definition.

The following table lists the attributes that an element of type `thread-pool-definition` can have:

Name	Type	Description	Required
name	push:string	Name of the thread pool definition.	true

The following table lists the elements that an element of type `thread-pool-definition` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
core-size	push:positiveInt	The core number of threads to have running in the thread pool. Whenever a thread is required a new thread is created until this number is reached even if there are idle threads already in the pool.	1	1
max-size	push:positiveInt	The maximum number of threads that can be created in the thread pool before tasks are queued. Such threads are released immediately after execution.	1	1
queue-size	push:positiveInt	The thread pool queue size. When the max-size is reached, tasks are queued. If the value is 0, the queue is unbounded. If the value is not 0, it must be at least 10.	1	1
keep-alive	push:millis	The time to keep inactive threads alive for. This does not apply to core threads. If this value is not specified, a default of 0 is used.	0	1
priority	push:threadPriority	This is the priority at which the threads run. If this value is not specified, a default of 5 is used.	0	1
thread-pool-listener	thread-pool-listener	Thread pool listener details (optional)	0	1

Name	Type	Description	Minimu occurre	Maximu occuren
rejection-handler-class	push:string	The name of a class implementing the ThreadPoolRejectionHandler interface which is called if a task cannot be executed by the Thread Pool. If this value is not specified, a default rejection policy is used so that rejected tasks are executed in the calling thread. The default rejection policy is implemented by the class com.pushtechnology.diffusion.api.threads.ThreadService.CallerRunsReject A thread is rejected if all the threads are in use and the queue is full.	0	1

thread-pool-listener

Thread pool listener details.

The following table lists the elements that an element of type `thread-pool-listener` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
queue-notification-handler-class	push:string	The name of a class implementing the ThreadPoolNotificationHandler interface which is instantiated to handle notifications on the thread pool.	1	1
queue-upper-threshold	push:percent	The size of the thread pool queue at which the notification handler is called on the queueUpperThresholdReached method. The method is called once only until the queue size drops below the specified lower threshold.	1	1
queue-lower-threshold	push:percent	The size of the thread pool queue at which the notification handler is called on the queueLowerThresholdReached method if the upper threshold has previously been breached.	1	1

whois

WhoIs service details.

The following table lists the elements that an element of type `whois` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
provider	push:string	Name of the WhoIs provider class that must be on the classpath and must implement the API class WhoIsProvider. If a provider is not specified, WhoIsDefaultProvider is used.	0	1
threads	push:int	The number of background threads that process WhoIs resolver requests. If a value is not specified, a default of 2 is used. If the value is set to 0, the service is not started.	0	1

Name	Type	Description	Minimu occurre	Maximu occuren
host	push:string	The hostname of a WhoIs provider that adheres to the RFC3912 WhoIs protocol. If a hostname is not specified, a default of "whois.ripe.net" is used.	0	1
port	push:port	The port number that the WhoIs provider listens on. If a value is not specified, the normal value of 43 is used.	0	1
whois-cache	whois-cache	Details of the WhoIs service cache that is used to cache WhoIs lookup results. If a value is not specified, the default values are used.	0	1

whois-cache

Details of the WhoIs service cache that is used to cache WhoIs lookup results.

The following table lists the elements that an element of type `whois-cache` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
maximum	push:int	The maximum size of the WhoIs cache. When the cache size exceeds this number it is tidied. A value of 0 means the cache grows indefinitely unless entries are removed because they have exceeded their retention time. If a value is not specified, a default of 1000 is used.	0	1
retention	push:millis	The time for which WhoIs cache entries are retained before being deleted. A value of 0 means entries are retained indefinitely or until the cache reaches its maximum size. If a value is not specified, a default of 0 is used.	0	1
tidy-interval	push:millis	The interval at which the Whois cache tidier task checks if any cache entries have passed their retention time or if the cache has exceeded its maximum size. This is ignored if both maximum and retention are 0. If a value is not specified, a default of 1 minute is used.	0	1

auto-deployment

Auto deployment details.

The following table lists the elements that an element of type `auto-deployment` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
directory	push:string	The name of the automatic deployment directory.	1	1

Name	Type	Description	Minimu occurre	Maximu occuren
scan-frequency	push:millis	The frequency at which the deployment directory is scanned for new deployments. If a value is not specified, a default of 5 seconds is used.	0	1

geo-ip

GeoIP details.

The following table lists the attributes that an element of type `geo-ip` can have:

Name	Type	Description	Required
enabled	push:boolean	Set to true to enable GeoIP lookup. This needs to be set to true if you are going to use connection or subscription validation policies. If a value is not specified, a default of true is used.	false

The following table lists the elements that an element of type `geo-ip` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
file-name	push:string	The name of the Maxmind GeoCityIP city file. If a value is not specified, a default of "../data/GeoLiteCity.dat" is used.	0	1

usr-lib

A list of user libraries from which user code is loaded.

The following table lists the elements that an element of type `usr-lib` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
directory	push:string	Directory to load classes from. When the server starts, this folder is traversed, including subdirectories and all jars or zip files added to the class loader.	1	unbounded

Related Links

Authentication handlers on page 418
You can implement authentication handlers that authenticate client connections to the Diffusion server or perform an action that requires authentication.

connectors

Connectors.xml - defines the connectors required.

connectors

Connectors

The following table lists the elements that an element of type `connectors` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
connector	connector	Connector definition	0	unbounded

connector

Connector definition

The following table lists the attributes that an element of type `connector` can have:

Name	Type	Description	Required
name	push:string	The connector name	true

The following table lists the elements that an element of type `connector` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
type	connectorType	The type of connection supported. By default 'all' types are supported but the connector can be restricted to one of the following specific types - 'client' (Clients only), 'event' (Event Publishers only), or 'policy' (Policy File Requests only).	0	1
api-type	connectorApiType	This setting constrains the API that can be used with this connector. The allowed values are 'all', 'classic', and 'unified'. The value 'unified' indicates that clients must use the Unified API. The value 'classic' indicates that clients must use the Classic API. The value 'all' indicates that clients can use either API. The default value is 'classic'.	0	1
host	push:string	The name or the IP address that the connector binds to. This is optional.	0	1
port	push:port	The port on which the connector accepts connections.	1	1
acceptors	push:positiveNonZeroInt	The number of acceptors to run for this connector. If this value is not specified, a default of 2 is used.	0	1
backlog	push:positiveNonZeroInt	The maximum queue length for incoming clients. If a connection indication arrives when the queue is full, the connection is refused. If a value is not specified, a default of 1000 is used.	0	1
socket-conditioning	socket-conditioning	Describes the properties associated with TCP socket connections.	1	1
web-server	push:string	If this connector is required to serve HTTP requests, this element specifies a web-server entry in WebServer.xml. If a value is not specified, the connector cannot serve HTTP requests.	0	1
policy-file	push:string	The location/name of the policy file if this connector is required to act as a policy file server (type='all' or 'policy').	0	1

Name	Type	Description	Minimu occurre	Maximu occuren
validation-policy-file	push:string	The location/name of a connection validation policy file to use for this connector. Applies only to type 'all' or 'client'.	0	1
key-store	key-store-definition	Keystore details for any connector that is to support secure (SSL) connections. If this is not specified, SSL connections are not supported.	0	1
queue-definition	push:string	An optional queue definition to use for this connector. This applies only to connectors of type 'all' or 'client'. The definition must exist in Server.xml. If this is not specified, the default queue definition specified in Server.xml is used.	0	1
reconnect	reconnect	Optional reconnection properties which apply only to connectors that accept 'client' connections. If this is not specified, reconnection of client connections is not supported.	0	1
ignore-errors-from	ignore-errors-from	Specifies addresses from which connection errors can be ignored. This is useful for masking errors that might be reported due to the connector port being pinged by some known external entity.	0	1
thread-pool-definition	push:string	Optionally, this can be used to specify a thread pool definition to be used for this connector to create its own inbound thread pool. If specified, the thread pool definition must exist in Server.xml. If a value is not specified, the default inbound thread pool is used.	0	1
system-ping-frequency	push:millis	This indicates the interval at which clients are pinged by the server to ensure that they are still connected. If a response is not received from the client before the expiry of another interval period, the client is assumed to be disconnected. If this is not specified or a value of 0 is supplied, clients are not automatically pinged.	0	1
fetch-policy	fetch-policy	Specifies a policy for batching fetch requests. If a value is not specified, no policy is applied and fetches are not batched.	0	1

socket-conditioning

Describes properties associated with TCP socket connections.

The following table lists the elements that an element of type `socket-conditioning` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
input-buffer-size	push:bytes	Specifies the size of the socket input buffer to use for each connection. This must be large enough to accomodate the largest inbound message expected. If a value is not specified, a default of 4k is used.	0	1
output-buffer-size	push:bytes	This value specifies the size of the output buffer to use for each connection. This must be large enough to accomodate the largest message to be sent. Messages are 'batched' into this buffer and so the larger the buffer, the more messages can be sent in a single write. If a value is not specified, a default of 64k is used.	0	1
keep-alive	push:boolean	This enables or disables TCP keep-alive. If a value is not specified, a default of true is used.	0	1
no-delay	push:boolean	This enables or disables TCP_NODELAY (disable/enable Nagle's algorithm). If a value is not specified, a default of true is used.	0	1
reuse-address	push:boolean	When a TCP connection is closed the connection can remain in a timeout state for a period of time after the connection is closed (typically known as the TIME_WAIT state or 2MSL wait state). For applications using a well-known socket address or port, it might not be possible to bind a socket to the required SocketAddress if there is a connection in the timeout state involving the socket address or port. Enabling this feature allows the socket to be bound even though a previous connection is in a timeout state. If this value is not specified, the feature is enabled.	0	1

reconnect

Reconnect properties.

The following table lists the elements that an element of type `reconnect` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
keep-alive	push:millis	This specifies the period within which a disconnected client can reconnect to the same client session. Messages for the client continue to be queued during this period.	1	1
max-depth	push:positiveInt	As messages continue to be queued for a client whilst it is disconnected, this enables you to specify a larger maximum queue size that is used during the period that the client is disconnected. When the client reconnects, the maximum	0	1

Name	Type	Description	Minimu occurre	Maximu occuren
		reverts back to its previous size (once any backlog had been cleared). If the specified size is not greater than the current maximum size, this has no effect. If this value is not specified, a default of 0 is used which means that no attempt is made to extend the queue size when a client is disconnected.		

key-store-definition

The keystore definition that allows SSL connection to a connector.

The following table lists the attributes that an element of type `key-store-definition` can have:

Name	Type	Description	Required
mandatory	push:boolean	If this is set to true, all connections must use this keystore and SSL connection is mandatory. If a value is not specified, a default of false is used, meaning that the connector accepts either SSL or non-SSL connections.	false

The following table lists the elements that an element of type `key-store-definition` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
file	push:string	The keystore file path.	1	1
password	push:string	The password for the keystore.	1	1

ignore-errors-from

Some external monitors cause the Diffusion server to log errors, as it is not a valid Diffusion connection. Adding the remote IP address to this list ensure that the errors are not logged.

The following table lists the elements that an element of type `ignore-errors-from` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
ip-address	push:string	An IP address or unknown if the remote IP address is being masked.	1	unbounded

fetch-policy

This is the policy for batching fetch requests. This can be used when fetches on topic sets might be large and lead to an excessive number of fetch reply messages being queued for a client at one time. The policy can define that the replies are sent in periodic batches to allow the client time to process them and prevent client queues filling.

The following table lists the elements that an element of type `fetch-policy` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
batch-size	push:positiveInt	Specifies the maximum number of fetch reply messages to send per batch. If this is set to 0, no batching occurs.	1	1

Name	Type	Description	Minimu occurre	Maximu occuren
delay	push:millis	Specifies the time period between submissions of batches. If a batch size is specified, this must be a positive value.	1	1

publishers

Publishers.xml - defines publishers.

publishers

The set of publishers that the Diffusion server is aware of at startup.

The following table lists the elements that an element of type `publishers` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
publisher	publisher	A publisher definition.	0	unbounded

publisher

A publisher definition.

The following table lists the attributes that an element of type `publisher` can have:

Name	Type	Description	Required
name	push:string	The publisher name.	true

The following table lists the elements that an element of type `publisher` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
topics	push:string	An optional, comma-separated list of topic names specifying topics to be automatically created for the publisher as it is started. This technique does not allow for topics to be set up with data and so it is more usual to define the topics you require in the initialLoad method of the Publisher. This property remains mostly for backwards compatibility.	0	1
class	push:string	The full class name of a Java class that implements the publisher. This class must extend the Java API Publisher class and provide implementations of methods as required. The class file must be available on the classpath of the Diffusion server (or in the configured usr-lib or ext folder).	1	1
enabled	push:boolean	By default, the publisher is loaded as the server starts. By setting this to false, the publisher is not loaded.	0	1
start	push:boolean	By default, the publisher is started after it is loaded. By specifying this as false, the publisher can be loaded but not started and then can be started later using JMX.	0	1

Name	Type	Description	Minimu occurre	Maximu occuren
topic-aliasing	push:boolean	Specifies whether topic aliasing is turned on for all topics created by the publisher. If the value is true, a short topic alias is transmitted in delta messages instead of the full topic name. By default, this is true, but because there are certain limitations when using topic aliasing there might be situations where you might want to turn it off.	0	1
ack-timeout	push:millis	This specifies the default ACK (message acknowledgment) timeout value (in milliseconds) to use for messages sent from the publisher that require acknowledgment and do not have a timeout explicitly specified. If a value is not specified, a default of 1s is used.	0	1
auto-ack	push:boolean	Indicates whether to automatically acknowledge messages sent from clients to the publisher requiring acknowledgment. By default, this is false so messages requiring acknowledgment must be manually acknowledged by the publisher.	0	1
subscription-policy-file	push:string	Path of a subscription validation policy file. If this value is specified, the file is used to validate client subscriptions to topics owned by the publisher.	0	1
stop-server-if-not-loaded	push:boolean	If this is set to true and the publisher fails to load, the Diffusion server stops. By default, this is false.	0	1
log-level	push:log-level	Specifies the log level for the publisher. If this value is not specified, the publisher logs at the default log level.	0	1
server	server	A specification of a server that is automatically connected to by the publisher when it starts.	0	unbounded
web-server	web-server	If the publisher has associated web content, it can be deployed with the publisher by specifying this property.	0	1
launch	launch	Launch detail describes how the publisher might be accessed externally, if it has an associated webpage.	0	unbounded
property	property	A property available to the publisher. This can be used to configure publisher-specific variables or parameters.	0	unbounded

launch

Launch detail.

The following table lists the attributes that an element of type `launch` can have:

Name	Type	Description	Required
name	push:string	The launcher name.	true
category	push:string	An optional category to which this launcher belongs. For example, "demo" for the Diffusion demo landing page.	false

The following table lists the elements that an element of type `launch` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
description	push:string	A short description of this launcher.	0	1
url	push:string	The URL at which a webpage associated with this publisher can be found.	1	1
icon	push:string	A URL or path at which an icon representing this launcher can be reached.	0	1

property

A publisher property.

The following table lists the attributes that an element of type `property` can have:

Name	Type	Description	Required
name	push:string	The property value	true
type	push:string	An optional property type. Usage of this is implementation specific.	false

credentials

Credentials for server connection.

The following table lists the elements that an element of type `credentials` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
username	push:string	User name.	0	1
password	push:string	Password.	0	1

server

The following table lists the attributes that an element of type `server` can have:

Name	Type	Description	Required
name	push:string	Server definition name.	true

The following table lists the elements that an element of type `server` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
host	push:string	The host name or IP address of the server.	1	1
port	push:port	The port number that the server is listening on for publisher client connections from other publishers.	1	1

Name	Type	Description	Minimu occurre	Maximu occuren
ssl	push:boolean	If this value is true, the connection to the server is a secure connection over SSL. In this case the specified port must represent an SSL client connector at the server. The keystore properties must also be supplied for secure connections. By default, this is false.	0	1
keystore-file-location	push:string	The path of the keystore file defining the SSL context. This is ignored if ssl=false, but mandatory if it is true.	0	1
keystore-password	push:string	The keystore password. This is ignored if ssl=false, but mandatory if it is true.	0	1
input-buffer-size	push:bytes	Specifies the size of the input buffer to use for the connection with the server. This is used to receive messages from the server. Set this to the same size as the output buffer used at the server.	1	1
output-buffer-size	push:bytes	The size of the output buffer to use for the connection with the server. This is used to send messages to the server. Set this to the same size as the input buffer used by the server.	1	1
fail-policy	push:string	This specifies what happens if the publisher fails to connect to the server. 'default' means that if unable to connect, no action is taken and it is the publisher's responsibility to handle this. 'close' means that if unable to connect to the server, the publisher closes. 'retry' means that if unable to connect, the connection is automatically retried at intervals as specified by the retry-interval property.	1	1
retry-interval	push:millis	If the fail-policy for a server is 'retry', this is the interval at which the connection to the server is retried. If this value is not specified, a default of 5s is used.	0	1
credentials	credentials	Credentials to use for the server connection. If this value is not specified, no credentials are passed on connection.	0	1
queue-definition	push:string	Optional outbound queue definition for this server connection. The definition must exist in Server.xml. This defines the queue to use for outbound messages from the publisher to the server. If this value is not specified, the default queue definition in Server.xml is used.	0	1

web-server

A web server definition.

The following table lists the elements that an element of type `web-server` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
virtual-host	push:string	The name of the virtual host to deploy to. If this value is not supplied, default-files-default is used.	0	1
alias-file	push:string	The alias file to use for this publisher	1	1

web-servers

WebServer.xml - definitions of one or more web servers.

web-servers

Definitions of one or more web servers.

The following table lists the elements that an element of type `web-servers` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
web-server	web-server	Web server definition.	0	unbounded

web-server

Web server definition.

The following table lists the attributes that an element of type `web-server` can have:

Name	Type	Description	Required
name	push:string	Name of the web server definition.	true

The following table lists the elements that an element of type `web-server` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
client-service	client-service	Optional client service.	0	1
http-service	http-service	HTTP service.	0	unbounded
file-service	file-service	Optional file service.	0	1

virtual-host

Virtual host definiton.

The following table lists the attributes that an element of type `virtual-host` can have:

Name	Type	Description	Required
name	push:string	Virtual host name.	true
debug	push:boolean	Debug flag. Set to true for debugging. Default is false.	false

The following table lists the elements that an element of type `virtual-host` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
host	push:string	Specifies the host which the virtual host is to serve, for example, download.pushtechnology.com or * for all.	1	1

Name	Type	Description	Minimu occurre	Maximu occuren
document-root	push:string	The physical directory for this virtual host.	1	1
home-page	push:string	The default home page. This file is used with directory browsing.	1	1
error-page	push:string	This is used to control the 404 response. The server looks for one of these files in the directory of the request. If the file does not exist, it looks for this file in the virtual directory. If the file is not supplied or the file does not exist, a standard 404 response HTML document is sent.	0	1
static	push:boolean	If this is set to true, after loading the resource once, the file system is not checked again. This improves performance for simple static usage. By default this is false.	0	1
minify	push:boolean	Set to true to minify the html. This happens before the file is compressed. By default this is false.	0	1
cache	cache	The virtual host cache configuration.	1	1
compression-threshold	push:bytes	All HTTP responses over this size are compressed. If not specified, a default value of 512 is used.	0	1
alias-file	push:string	Optionally specifies an alias file. This allows for URL aliasing if required.	0	1
realms	realms	Virtual host realms.	0	1

realms

Virtual host realms.

The following table lists the elements that an element of type `realms` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
realm	realm	A virtual host realm.	0	unbounded

realm

A virtual host realm.

The following table lists the attributes that an element of type `realm` can have:

Name	Type	Description	Required
name	push:string	Virtual host realm name.	true
path	push:string	Virtual host realm path.	true

The following table lists the elements that an element of type `realm` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
users	users	Virtual host realm users.	0	1

users

Virtual host realm users.

The following table lists the elements that an element of type `users` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
user	user	Virtual host realm user.	1	unbounded

user

Virtual host realm user.

The following table lists the attributes that an element of type `user` can have:

Name	Type	Description	Required
name	push:string	Virtual host realm user name.	true
password	push:string	Virtual host realm user password.	true

cache

Virtual host cache.

The following table lists the attributes that an element of type `cache` can have:

Name	Type	Description	Required
debug	push:boolean	Set true to debug the cache. If a value is not specified, a default of false is used.	false

The following table lists the elements that an element of type `cache` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
file-size-limit	push:bytes	If the file to be served is over this size, do not cache the entire contents, but map the file instead. If a size is not specified, a default value of 1m is used.	0	1
cache-size-limit	push:bytes	Total size of the cache for this web server definition. If a size is not specified, a default value of 10m is used.	0	1
file-life-time	push:millis	If the file has not been accessed within the time specified, remove the entry from the cache. If a time is not specified, a default value of 1d is used.	0	1

http-service

HTTP service.

The following table lists the attributes that an element of type `http-service` can have:

Name	Type	Description	Required
name	push:string	HTTP service name.	true
debug	push:boolean	Set true to debug the HTTP service. If a value is not specified, a default of false is used.	false

The following table lists the elements that an element of type `http-service` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
class	push:string	The user HTTP service class name. This class must implement the HTTPServiceHandler interface in the web server API.	1	1
url-pattern	push:string	The pattern that the URL must match for this service to be invoked.	1	1
log	push:string	An optional log file can be specified and, if so, HTTP access can be logged. The log definition must exist in Logs.xml.	0	1
max-inbound-request-size	push:bytes	The maximum number of bytes that the HTTP request can have. If this is not specified, a default of the maximum message size is used.	0	1
property	property	HTTP service property.	0	unbounded

property

A property.

The following table lists the attributes that an element of type `property` can have:

Name	Type	Description	Required
name	push:string	Property name.	true
type	push:string	Optional property type.	false

file-service

File service.

The following table lists the attributes that an element of type `file-service` can have:

Name	Type	Description	Required
name	push:string	File service name.	true

The following table lists the elements that an element of type `file-service` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
virtual-host	virtual-host	Virtual host.	1	unbounded
write-timeout	push:millis	Write timeout for serving files. This does not affect HTTP clients. If a value is not specified, a default value of 3s is used.	0	1

client-service

Client service.

The following table lists the attributes that an element of type `client-service` can have:

Name	Type	Description	Required
name	push:string	Client service name.	true

Name	Type	Description	Required
debug	push:boolean	Set true to debug the client service. If a value is not specified, a default of false is used.	false

The following table lists the elements that an element of type `client-service` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
message-sequence-timeout	push:millis	This is used with HTTP clients to indicate how long to wait for a missing message in a sequence of messages before assuming it has been lost and closing the client connection. If a value is not specified, a default of 2s is used.	0	1
websocket-origin	push:string	This is used to control access from client web socket to Diffusion. This is a regular expression pattern that matches the origin of the request. A value of ".*" matches anything, so all requests are allowed. If this is not specified, the service is unable to handle web socket requests.	0	1
cors-origin	push:string	This is used to control access from client web (XHR) to Diffusion. This element will enable Cross Origin Resource Sharing (CORS). This is a regular expression pattern that matches the origin of the request. A value of ".*" matches anything, so all requests are allowed. If a value is not specified, the service cannot handle CORS requests.	0	1
websocket-secure-response	push:boolean	Indicates that the websocket response states that it is from a secure connection, when it is not. Use this option if SSL off-loading is enabled on a load-balancer. If a value is not specified, a default of false is used.	0	1
close-callback-requests	push:boolean	For Diffusion client requests this specifies whether to obey the keep-alive header or close all requests. If this is set to true, all requests are closed. If a value is not specified, a default of false is used.	0	1
compression-threshold	push:bytes	Enable compression for HTTP client responses over this size. If a value is not specified, a default of 512 bytes is used.	0	1
max-inbound-request-size	push:bytes	The maximum number of bytes that the HTTP request can have. If a value is not specified, a default of the maximum message size is used.	0	1
comet-bytes-before-new-poll	push:bytes	This parameter enables you to specify the number of bytes after which a Comet connection is forced to re-establish itself. This can help to reduce the potential for memory leaks in the browser due to the long-lived nature of a Comet connection,	0	1

Name	Type	Description	Minimu occurre	Maximu occuren
		at the expense of degraded performance due to more frequent HTTP handshakes. If this is not specified, a default value of 30 kilobytes is used.		
comet-initial-message-padding	push:bytes	Some browsers do not pass on data received through a Comet connection until a minimum number of bytes have been received. This means that in the case where an initial topic load message is small, the client might never receive it. To work around this restriction, you can set a value here which ensures that the first message received is padded with extra bytes that are automatically discarded by the client library. If a value is not specified, a default of 1k is used.	0	1

logs

Logs.xml - Properties defining logging options.

logs

Properties defining logging options.

The following table lists the elements that an element of type `logs` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
console-log-level	push:log-level	The log level to start console logging at. Can be SEVERE, WARNING, ADVICE, INFO, FINE, or FINEST. If a value is not specified, a default of INFO is used.	0	1
log-message-data	push:boolean	Indicates whether the data part of messages is logged as part of routine diagnostic message logging (FINE and FINEST levels). If this is false, credentials message headers are also hidden. If a value is not specified, a default of true is used.	0	1
server-log	push:string	The log to use for the server. This must specify the name of a configured log definition.	1	1
default-log-directory	push:string	The default log folder for all logs, although this can be over-ridden for each log.	1	1
async-logging	push:boolean	Indicates whether logging is asynchronous. Asynchronous logging is performed by a separate thread as opposed to being performed in-line by the logging thread. This is normally set to true for performance reasons, but asynchronous logging might cause problems in some OS environments. This element provides the option to turn	0	1

Name	Type	Description	Minimu occurre	Maximu occuren
		asynchronous logging off, if so advised. If a value is not specified, a default of true is used.		
logging-queue-size	push:positiveInt	The size of the asynchronous logging queue. In normal cases, leave this value at the default value of 128k entries.	0	1
thread-name-logging	push:boolean	Indicates whether the thread name is logged with messages. If this is not specified, thread names are logged.	0	1
log	log	A log definition.	0	unbounded

log

A log definition.

The following table lists the attributes that an element of type `log` can have:

Name	Type	Description	Required
name		Name of the log definition	true
rotation-period	push:positiveNonZeroInt	A time period that the log exists for before being rotated. This is a positive non-zero integer, with unit specified by rotation-unit.	false
rotation-unit	push:timeunit	A time unit to specify the unit used alongside rotation-period. This can be "day(s)", "hour(s)", "minute(s)".	false

The following table lists the elements that an element of type `log` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
log-directory	push:string	The name of the directory to which this log file is written. If a value of not specified, the default-log-directory is used.	0	1
file-pattern	push:string	This is used to specify the name of the log file. The following values can be used within the pattern. "/" - the local pathname separator. "%t" - the system temporary directory. "%g" - the generation number to distinguish rotated logs. "%h" - the value of the "user.home" system property. "%u" - a unique number to resolve conflicts. "%s" - the system type - for example, 'Diffusion'. "%n" - the system name as defined in Server.xml. "%d" - the date as specified in diffusion.properties (date.format), this is included when using daily rotation. "%%" - translates to a single percent sign "%". If a log file name is not specified, a default of "%s.log" is used.	0	1

Name	Type	Description	Minimu occurre	Maximu occuren
level	push:log-level	Specifies the starting log level. This can be SEVERE, WARNING, ADVICE, INFO, FINE, or FINEST. If a value is not specified, a default of ADVICE is used.	0	1
xml-format	push:boolean	Indicates whether the log file is output in XML format. If a value is not specified, a default of false is used.	0	1
file-limit	push:bytes	Specifies an approximate maximum amount to write (in bytes) to any one log file. If this is zero, there is no limit. If a value is not specified, a default of 0 is used.	0	1
file-append	push:boolean	Specifies whether log records are appended to existing log files. If a value is not specified, a default of false is used and log files are overwritten.	0	1
file-count	push:positiveNonZeroInt	Specifies the number of log files to use. Must be at least 1. If a value is not specified, a default of 1 is used.	0	1
rotate-daily	push:boolean	Indicates whether the log is to rotate on a daily basis. This is superceded by rotation-period.	0	1

management

Management.xml - specifies system management properties.

management

The management information

The following table lists the attributes that an element of type `management` can have:

Name	Type	Description	Required
enabled	push:boolean	Specifies if JMXRMI services are started, making JMX remotely available.	true

The following table lists the elements that an element of type `management` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
host	push:string	The management host (RMI registry host). If a value is not specified, a default of "localhost" is used.	0	1
registry-port	push:port	The RMI registry port. If a value is not specified, a default of 1099 is used.	0	1
connection-port	push:port	The JMXRMI connection port. If a value is not specified, a default of 1100 is used. This relates to the normally ephemeral port employed by JMXRMI. This is normally useful only when configuring firewalls.	0	1

Name	Type	Description	Minimu occurre	Maximu occuren
register-topics	push:boolean	Vestigial setting with zero effect. Superseded by features in Statistics.xml.	0	1
users	users	The management users.	1	1

users

Management users.

The following table lists the elements that an element of type `users` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
user	user	A user that can use the JMX interface.	1	unbounded

user

Management user.

The following table lists the elements that an element of type `user` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
name	push:string	User name for JMX credentials.	1	1
password	push:string	Password for JMX credentials.	1	1
read-only	push:boolean	Specify if the user has read-only access. If this value is false, the user has admin access.	1	1

replication

Replication.xml - defines the replication performed by the server.

replication

Properties defining replication.

The following table lists the attributes that an element of type `replication` can have:

Name	Type	Description	Required
enabled	push:boolean	Specifies whether replication is enabled for this server.	true

The following table lists the elements that an element of type `replication` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
provider	push:string	The type of replication provider to use to replicate the data. Currently only Hazelcast is supported.	1	1
sessionReplication	sessionReplication	The definition for session replication	1	1
topicReplication	topicReplication	The definition for topic replication	1	1

sessionReplication

Properties defining session replication.

The following table lists the attributes that an element of type `sessionReplication` can have:

Name	Type	Description	Required
enabled	push:boolean	Specifies whether session replication is enabled for this server.	true

topicReplication

Properties defining topic replication.

The following table lists the attributes that an element of type `topicReplication` can have:

Name	Type	Description	Required
enabled	push:boolean	Specifies whether topic replication is enabled for this server.	true

The following table lists the elements that an element of type `topicReplication` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
topics	topics	The topics that are configured to use replication.	1	1

topics

Properties defining the topics to replicate.

The following table lists the elements that an element of type `topics` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
topicPath	push:string	A topic path that identifies the root of a tree that will be replicated by this server.	0	unbounded

Related Links

Session replication on page 67
You can use session replication to ensure that if a client connection fails over from one server to another the state of the client session is maintained.

Topic replication on page 69
You can use topic replication to ensure that the structure of the topic tree, topic definitions, and topic data are synchronized between servers.

Failover of active update sources on page 70
You can use failover of active update sources to ensure that when a server that is the active update source for a section of the topic tree becomes unavailable, another server is assigned to be the active update source for that section of the topic tree. Failover of active update sources is enabled for any sections of the topic tree that have topic replication enabled.

Configuring replication on page 72

You can configure replication by editing the `etc/Replication.xml` files of your Diffusion servers.

statistics

Statistics.xml - properties defining statistics collection. The statistics are broken into sections: client, topic, server and publisher.

statistics

Properties defining statistics collection.

The following table lists the attributes that an element of type `statistics` can have:

Name	Type	Description	Required
enabled	push:boolean	A global switch to toggle collection of all statistics.	true

The following table lists the elements that an element of type `statistics` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
client-statistics	client-statistics	Optional client statistics: configures Diffusion to periodically output client statistics to a log file defined in Logs.xml It gives a count of all of the different client types. Each counter is reset according to the configured frequency.	0	1
topic-statistics	topic-statistics	Optional topic statistics.	0	1
server-statistics	server-statistics	Optional server statistics.	0	1
publisher-statistics	publisher-statistics	Optional publisher statistics.	0	1
reporters	reporters	Optional set of StatisticsReporters to be loaded with Diffusion, which are registered with the internal StatisticsService and used to generate output.	0	1

client-statistics

The following table lists the attributes that an element of type `client-statistics` can have:

Name	Type	Description	Required
enabled	push:boolean	Specifies if aggregate client statistics are enabled.	true

The following table lists the elements that an element of type `client-statistics` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
log-name	push:string	Definition of the log in Logs.xml.	0	1
output-frequency	push:millis	Specifies the output frequency of the log. There is one entry per specified interval. If this is not specified, a default of 1h is used.	0	1

Name	Type	Description	Minimu occurre	Maximu occuren
reset-frequency	push:millis	Specifies when the counters are reset. Zero specifies that the counters are never reset. If this is not specified, a default of 1h is used.	0	1
monitor-instances	push:boolean	Specifies if individual client statistics are enabled.	0	1

topic-statistics

The following table lists the attributes that an element of type `topic-statistics` can have:

Name	Type	Description	Required
enabled	push:boolean	Specifies if aggregate topic statistics are enabled.	true

The following table lists the elements that an element of type `topic-statistics` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
monitor-instances	push:boolean	Specifies if individual topic statistics are enabled.	0	1

publisher-statistics

The following table lists the attributes that an element of type `publisher-statistics` can have:

Name	Type	Description	Required
enabled	push:boolean	Specifies if aggregate publisher statistics are enabled.	true

The following table lists the elements that an element of type `publisher-statistics` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
monitor-instances	push:boolean	Specifies if individual publisher statistics are enabled.	0	1

server-statistics

The following table lists the attributes that an element of type `server-statistics` can have:

Name	Type	Description	Required
enabled	push:boolean	Specifies whether to enable server statistics. This enables high-level aggregate statistics for the system.	true

The following table lists the elements that an element of type `server-statistics` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
monitor-instances	push:boolean	Specifies if individual event publisher statistics are enabled.	0	1

reporters

The set of StatisticsReporters that the Diffusion server is aware of at startup. Used to output the statistics gathered for clients, publishers, or topics.

The following table lists the elements that an element of type `reporters` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
reporter	reporter	A reporter definition.	0	unbounded

reporter

A StatisticsReporter definition.

The following table lists the attributes that an element of type `reporter` can have:

Name	Type	Description	Required
name	push:string	The reporter name.	true
enabled	push:boolean	Whether the reporter is enabled. If this is set to true, the reporter is automatically loaded when Diffusion starts. Otherwise, you must manually load the reporter config at run-time using the statistics API.	true

The following table lists the elements that an element of type `reporter` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
type	push:string	The type of reporter to be used. Currently options are: TOPIC - exposes metrics in the Diffusion topic tree; JMX - exposes metrics on the local JMX server.	1	1
property	property	A property available to the reporter. This can be used to configure reporter-specific variables or parameters.	0	unbounded

property

A StatisticsReporter property. Currently accepted values:
 'interval' - used by the topic reporter. Specifies an integer value used to set the period of update publishing.

The following table lists the attributes that an element of type `property` can have:

Name	Type	Description	Required
name	push:string	The property value	true
type	push:string	An optional property type. Usage of this is implementation specific.	false

connection-validation-policies

A connection validation policy file.

connection-validation-policies

Connection validation policies

The following table lists the elements that an element of type `connection-validation-policies` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
policy	policy	A connection validation policy.	0	unbounded

policy

A connection validation policy.

The following table lists the attributes that an element of type `policy` can have:

Name	Type	Description	Required
name	push:string	Each policy must be supplied with a unique name for easy reference.	true
type	push:string	The policy type should be either "blacklist" or "whitelist". A blacklist indicates that if any of the policy rules in this policy match the incoming connection, that connection is to be rejected. A whitelist requires that at least one policy rule matches for the connection to be accepted.	true
automatic	push:boolean	Policies which are set to automatic are applied by Diffusion and the publishers do not need to perform any checks themselves. If this attribute is set to false, the policy is not applied unless it is done so by the publisher. If a value is not specified, a default of true is used.	false

The following table lists the elements that an element of type `policy` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
addresses	addresses	Connection validation policy addresses. These are addresses that are blacklisted/ whitelisted.	0	1
locale	locale	Connection validation policy locale. This is locale details that are blacklisted/ whitelisted.	0	unbounded

addresses

The following table lists the elements that an element of type `addresses` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
address	push:string	An IP address (or regular expression) of a connecting client.	0	unbounded

Name	Type	Description	Minimu occurre	Maximu occuren
hostname	push:string	The hostname (or regular expression) of a connecting client.	0	unbounded
resolved-name	push:string	The resolved hostname (or regular expression) of a connecting client, as returned by the WhoIs service.	0	unbounded

locale

The following table lists the elements that an element of type `locale` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
country	push:string	The ISO country code of the connecting client, as returned by the WhoIs service.	0	1
language	push:string	The ISO language code of the connecting client, as returned by the WhoIs service.	0	1

subscription-validation-policies

A subscription validation policy file.

subscription-validation-policies

Subscription validation policies

The following table lists the elements that an element of type `subscription-validation-policies` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
topics	topics	A map of topics to policies.	0	1
policy	policy	A subscription validation policy.	0	unbounded

topics

A map of topics to policies.

The following table lists the elements that an element of type `topics` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
topic	topic	A topic to policy mapping.	0	unbounded

topic

The following table lists the attributes that an element of type `topic` can have:

Name	Type	Description	Required
policy	push:string	The name of the policy to apply to this topic.	true

policy

A subscription validation policy.

The following table lists the attributes that an element of type `policy` can have:

Name	Type	Description	Required
name	push:string	Each policy must be supplied with a unique name for easy reference.	true
type	push:string	The policy type is either "blacklist" or "whitelist". A blacklist indicates that if any of the policy rules in this policy match the incoming connection, that connection is to be rejected. A whitelist requires that at least one policy rule matches for the connection to be accepted.	true
automatic	push:boolean	Policies which are set to automatic are applied by Diffusion and the publishers do not need to perform any checks themselves. If this is set to false, the policy is not applied unless it is done by the publisher. If this value is not specified, a default of true is used.	false
validate-children	xsd:boolean	Controls whether to perform validation on child topics if the parent topic fails validation. If a value is not specified, a default of false is used.	false

The following table lists the elements that an element of type `policy` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
addresses	addresses	Subscription validation policy addresses. These are addresses that are blacklisted/ whitelisted.	0	1
locale	locale	Connection validation policy locale. This is locale details that are blacklisted/ whitelisted.	0	unbounded

addresses

The following table lists the elements that an element of type `addresses` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
address	push:string	An IP address (or regular expression) of a subscribing client.	0	unbounded
hostname	push:string	The hostname (or regular expression) of a subscribing client.	0	unbounded
resolved-name	push:string	The resolved hostname (or regular expression) of a subscribing client, as returned by the WhoIs service.	0	unbounded

locale

The following table lists the elements that an element of type `locale` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
country	push:string	The ISO country code of the subscribing client, as returned by the WhoIs service.	0	1
language	push:string	The ISO language code of the subscribing client, as returned by the WhoIs service.	0	1

env

Env.xml - environment variables used in configuration.

env

The following table lists the elements that an element of type `env` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
property	property	Environment variable value	0	unbounded

property

The following table lists the attributes that an element of type `property` can have:

Name	Type	Description	Required
name	xsd:token	Name of the environment variable.	true

aliases

Aliases.xml - properties that define the aliases used in a web server.

aliases

List of aliases

The following table lists the elements that an element of type `aliases` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
alias	alias	An alias definition	0	unbounded

alias

An alias definition

The following table lists the attributes that an element of type `alias` can have:

Name	Type	Description	Required
name	push:string	A name for the alias.	true

The following table lists the elements that an element of type `alias` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
source	push:string	The source URL, which can be expressed as a regular expression.	1	1
destination	push:string	The destination path.	1	1

mimes

Mimes.xml - mime types

mimes

The following table lists the elements that an element of type `mimes` can contain:

Name	Type	Description	Minimu occurre	Maximu occuren
mime	mime	Mime.	0	unbounded

mime

The following table lists the attributes that an element of type `mime` can have:

Name	Type	Description	Required
type	push:string	Mime type.	true
extension	push:string	Mime extension.	true

Additional XML files

The `etc` directory contains additional XML files. The format of these XML files is not defined by Push Technology.

The following XML files are included in the `etc` directory. Their use is described in other sections of this manual.

`crossdomain.xml`

`FlashMasterPolicy.xml`

`FlashPolicy.xml`

Use these files to grant a web client permission to handle data across multiple domains.

For more information, see Cross domain on page 112.

`JMSAdapter.xml`

Use this file to configure the JMS adapter.

For more information, see Configuring the JMS Adapter on page 488.

Programmatic configuration

An alternative to configuring a Diffusion server using XML property files is to instantiate a Diffusion server within a Java application and configure it programmatically before starting it.

Using this technique, you can do without XML property files altogether as every aspect of the XML configuration can also be supplied programmatically.

If desired, some properties can be loaded from XML files and some supplied programmatically or default properties can be bootstrapped from XML files and overridden programmatically before the server is started.

Most server properties can be configured only before the server is started. Instantiate the server within an application and configure before starting the server. However, certain configuration items (examples being conflation and connection policies) can be configured at any time during the life of the server. The API documentation makes it clear if a property can be changed at runtime.

Because the properties that can be set programmatically reflect those that can be set in XML this section does not describe the properties in detail. The XSD property descriptions or the API documentation for the configuration API can be consulted for full details.

As well as allowing configuration properties to be set the configuration API also allows all properties that can be configured to be read at runtime. So publisher code has direct access to all property settings.

Using the configuration API

General use

From within a Java application the root of the configuration tree can be obtained at any time using `ConfigManager.getConfig()`. This provides access to the general objects and can be used from within server-side or client-side code.

From within server-side code (for example, a publisher) the server configuration root can be obtained using `ConfigManager.getServerConfig()` which exposes all of the server side configuration also.

From the configuration root you can navigate to any subordinate configuration objects to view them or set their properties.

On the server side most properties cannot be changed after the server has started and they become locked so any attempt to change them results in an exception. Certain properties (such as conflation and connection policies) can be changed at runtime. The API documentation makes it clear which properties can be changed at runtime.

In client-side Java code the configuration does not become locked and can be changed at any time. However, some values are read at the start only. Ideally, set all properties before creating any client side objects.

For configuration objects which are optional but there can be many (multiplicity 0..n), there are appropriate add methods to add new objects. For example to add a publisher and set a property on it:

```
PublisherConfig publisher =
    ConfigManager.getServerConfig().addPublisher(
        "MyPublisher",
        "com.pub.MyPublisher");
publisher.setTopicAliasing(false);
```

In these cases there are also methods to obtain the full list (for example, `getPublishers()`) or to obtain a specific one by name (for example, `getPublisher("MyPublisher")`). In many cases there are also methods to remove an object.

> **Note:** When there must be at least one object (multiplicity 1..n), you must configure at least one. However, if a server is started with missing configuration of this kind, suitable defaults are normally created and a warning logged.

Single instance configuration objects (multiplicity 1..1) subordinate to the root can be obtained so that their properties can be changed (or read). So, for example the `Queues` object (an instance of `QueuesConfig`) can be obtained using the `getQueues()` method.

When a single configuration object is optional (multiplicity 0..1), the `get` method can return null if it has not been defined. In this case to set it the set method (as opposed to add) returns the object created. An example of this is the file service (`FileServiceConfig`) on a web server (`WebServerConfig`) as shown in the following example code:

```
ServerConfig config = ConfigManager.getServerConfig();
WebServerConfig webServer = config.addWebServer("MyWebServer");
FileServiceConfig fileService = webServer.setFileService("File Service");
```

Configuring a server

After instantiating a Diffusion server in Java the root of the server configuration tree can be obtained from the server object itself and configuration objects can be navigated to and changed as required before starting the server.

For example, the following code shows how to add a connector that accepts client connections on port 9090:

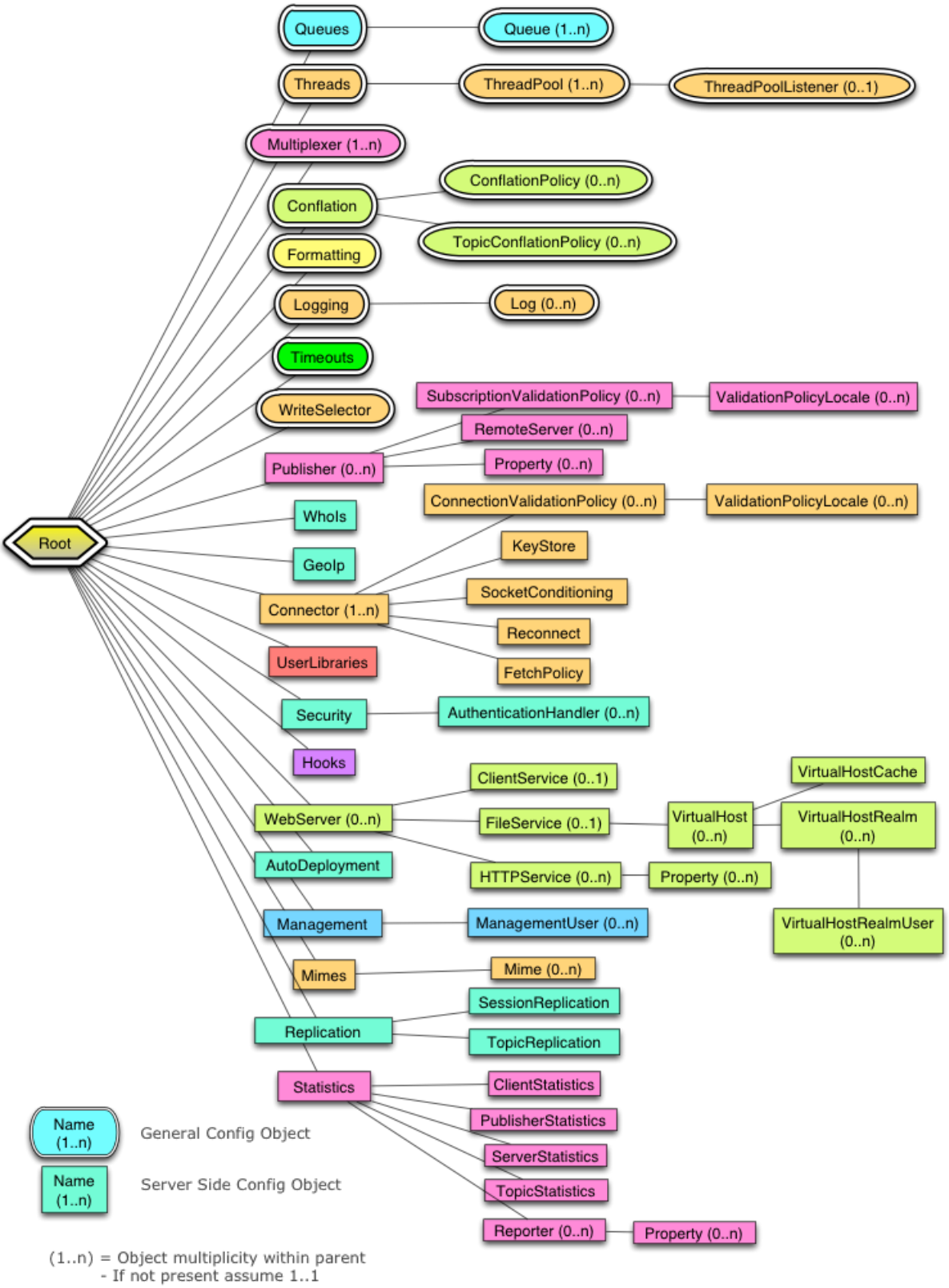
```
DiffusionServer server = new DiffusionServer();
ServerConfig config = server.getConfig();
ConnectorConfig connector = config.addConnector("Client Connector");
connector.setPort(9090);
connector.setType(Type.CLIENT);
server.start();
```

In reality, it is best to configure far more values. However, if any essential objects are omitted (such as queues), suitable defaults are created when the server starts and a warning is logged.

Configuration access from a publisher

Within a publisher the configuration object for the publisher itself can be obtained using the `getConfig` method which returns the publisher configuration (`PublisherConfig`) object.

The configuration tree



All general objects can be obtained by navigating from the root object obtained from `ConfigManager.getConfig()`.

Server-side objects can be reached only in a server environment using `ConfigManager.getServerConfig()`.

Chapter 17

Adapters

In this section:

- JMS Adapter

Adapters are connectors to external systems, providing a simplified bridge between data held in those systems and Diffusion.

JMS Adapter

The JMS Adapter for Diffusion allows Diffusion clients to transparently send and receive messages with topics and queues on a JMS server.

The adapter exposes JMS destinations in a Diffusion topic tree transparently, with no Java coding required, and allows Diffusion clients to both receive and send messages with a JMS server.

Mapping Diffusion topics to JMS topics and queues

Diffusion maintains a topic tree which maps to JMS topics and queues. Configure each JMS adapter with a root topic, for example `jms/activemq`, to partition itself from other JMS adapter instances. In `JMSAdapter.xml`, this is set using the `<root-topic>` element.

Underneath the root topic, there are four reserved sub-topics. Given a root topic `jms`, there are subtopics `topic`, `queue`, `tmp` and `reply`. The `tmp` subtopic also contains `topic` and `queue`, which map to JMS `TemporaryTopic` and `TemporaryQueue` destinations respectively.

The `reply` topic is reserved for subtopics associated with request-reply messages originating from another client of the JMS server. An example of how this is used is located in the examples below.

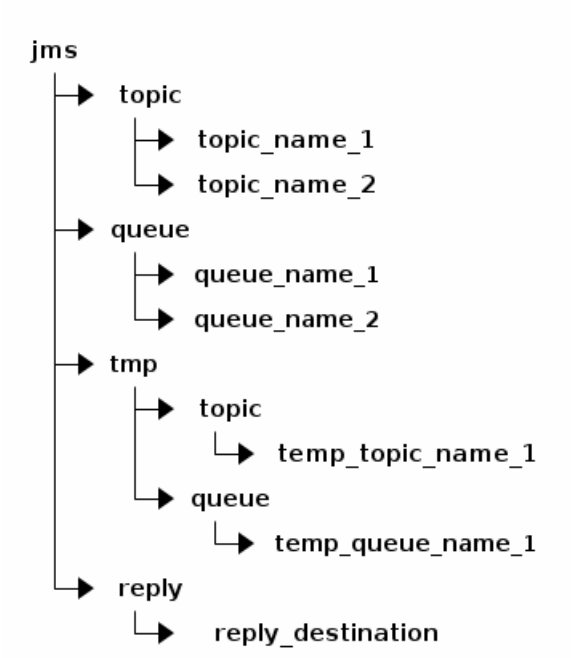


Figure 56: JMS adapter topic tree layout

When a client subscribes to the Diffusiontopic, `jms/activemq/topic/ABC` that has not previously been subscribed to, Diffusion attempts to transparently connect to the matching JMS topic and any messages that Diffusion receives from the JMS server are relayed to the Diffusion client. At present, you cannot subscribe to JMS topics using wildcards.

> **Note:** Diffusion topics are created dynamically and do not exist until a valid JMS subscription has been made and data received.

Mapping to JMS queues is performed in the same way. Delivery of messages to clients subscribing to the associated Diffusion topic is different, however, and in keeping with the delivery characteristics of JMS queues. When there are multiple Diffusion clients listening for data originating from the same JMS queue each message is delivered to at most one client. Depending on the configuration in `JMSAdapter.xml` the receiving client is chosen either randomly or based on the client with the fewest number of messages waiting for delivery.

Temporary topics and queues

A Diffusion client can request access to a JMS temporary topic (or queue). This is achieved in the same way as subscribing to a JMS destination except that it uses a different part of the Diffusion topic tree. A common use for temporary queues is to set up a return path for request-reply operations. In the topic tree, temporary topics exist as sub-topics under `jms/tmp/topic`.

Message headers

The JMS adapter copies the standard JMS headers and user-supplied headers when converting between JMS and Diffusion message types. In Diffusion, headers are logically grouped in pairs, where property n is the name, and n+1 is the value (where n is an even number).

In the case of the `JMSReplyTo` header, the related header `DiffusionReplyTo` is created. Mapping between a `JMSReplyTo` destination and a `DiffusionReplyTo` topic is handled transparently by the adapter. In most circumstances, a Diffusion client can ignore the `JMSReplyTo` header; it is forwarded to the client for completeness. Example usage of `DiffusionReplyTo` follows in the examples section.

Acknowledgment modes

The only acknowledgment mode supported is `AUTO_ACKNOWLEDGE`. This implies that when any message is received from the JMS server, an acknowledgment is sent from the JMS adapter to the JMS server. Client-level acknowledgments (`CLIENT_ACKNOWLEDGE`) originating from a Diffusion client are not supported.

Installing the JMS adapter

To use the JMS adapter for Diffusion, you need the following files:

Procedure

1. `jmsadapter.jar` in the Diffusion `CLASSPATH` (for example, in the `lib` directory).
2. The JMS library for your specific JMS vendor in the Diffusion `CLASSPATH`.
3. An entry in `etc/Publishers.xml` enabling the JMS adapter with a link to the `JMSAdapter.xml` file.
4. A correctly configured `JMSAdapter.xml` file. For validation purposes, the associated XML schema is located in the file `xsd/JMSAdapter.xsd` in the Diffusion installation directory.

Configuring the JMS Adapter

The configuration file for the JMS adapter is typically called `JMSAdapter.xml`, although you can override this by using a setting in `Publishers.xml`.

Procedure

1. Configure `Publishers.xml`

   Instantiate the JMS Adapter by enabling it in `etc/Publishers.xml`, for example:

   ```
   <publisher name="JMSAdapter">
     <class>com.pushtechnology.diffusion.adapters.jms.JMSAdapter</class>
     <enabled>true</enabled>
     <start>true</start>
     <property name="config.filename">../etc/JMSAdapter.xml</property>
   </publisher>
   ```

2. Configure `JMSAdapter.xml`

   A `JMSAdapter.xml` for ActiveMQ can look like this:

   ```
   <?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
   <jms-config>
   ```

```
  <binding>
    <env>
      <property name="java.naming.factory.initial">
        org.apache.activemq.jndi.ActiveMQInitialContextFactory
      </property>
      <property name="java.naming.provider.url">
 tcp://localhost:61616
      </property>
      <property name="java.naming.security.principal">
 jndi_username
      </property>
      <property name="java.naming.security.credentials">
 jndi_password
      </property>
    </env>
    <connection-factory name="ConnectionFactory">
      <credentials username="username" password="password"/>
      <reconnect>
        <max-reconnections>10</max-reconnections>
        <interval>5000</interval>
      </reconnect>
    </connection-factory>
    <root-topic>jms/activemq</root-topic>
    <priority low="3" high="7" />
    <queue-distribution-mode>SMALLEST_QUEUE</queue-distribution-mode>
  </binding>
  <mapping>
    <artefact-names jms=".$" diffusion="/~"/>
  </mapping>
</jms-config>
```

Similarly, a sample `JMSAdapter.xml` TIBCO Enterprise Message Service™ looks like this:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<jms-config>
  <binding>
    <env>
      <property name="java.naming.factory.initial">
        com.tibco.tibjms.naming.TibjmsInitialContextFactory
      </property>
      <property name="java.naming.provider.url">
        tcp://localhost:7222
      </property>
      <property name="java.naming.security.principal">
 jndi_username
      </property>
      <property name="java.naming.security.credentials">
 jndi_password
      </property>
    </env>
    <connection-factory name="ConnectionFactory">
      <credentials username="jms_username" password="jms_password"/>
      <reconnect>
 <max-reconnections>10</max-reconnections>
 <interval>5000</interval>
      </reconnect>
    </connection-factory>
    <root-topic>jms/tibco</root-topic>
    <priority low="3" high="7" />
    <queue-distribution-mode>SMALLEST_QUEUE</queue-distribution-mode>
  </binding>
  <mapping>
    <artefact-names jms=".$" diffusion="/~"/>
  </mapping>
</jms-config>
```

TIBCO Enterprise Message Service requires that a `ConnectionFactory` definition is provided in `factories.conf`, for example:

```
[ConnectionFactory]
  type = generic
  url = tcp://localhost:7222
  ssl_verify_host = disable
```

This has been tested with TIBCO Enterprise Message Service 7.0; see the TIBCO Enterprise Message Service documentation for further details.

For WebSphere® MQ v7.x, the binding section can look like this:

```
<env>
  <property
 name="java.naming.factory.initial">com.sun.jndi.fscontext.RefFSContextFactory</
property>
  <property name="java.naming.provider.url">file:///var/mqm/jndi</
property>
</env>
```

WebSphere MQ can include MQRFH2 headers in messages sent between JMS and non-JMS systems. For control over this behavior, set the property in `mq.target.client` in `Publishers.xml`; to disable the headers, set this value to `1`. For the default behavior, do not provide the property.

```
<property name="mq.target.client">1</property>
```

Some JMS vendors (for example, WebSphere MQ) require a JMS `Session` for each topic or queue subscription. The default configuration for the JMS Adapter does not allow for this, but you can enable it by setting the `use.global.session` property to `false` in `Publishers.xml`:

```
<property name="use.global.session">false</property>
```

Table 87: Properties that can be specified when configuring the JMS adapter

<env>	All properties within the `<env>` tag of `JMSAdapter.xml` are used when creating the `InitialContext` which is in turn used to create the connection to the JMS server.
<connection-factory>	This tag specifies the name of the connection factory to use. This varies between JMS vendors and the server configuration.
<credentials>	Optionally, specify a username and/or password which is used to create a JMS client connection. However, most JMS implementations are likely to have restricted access for anonymous clients or clients which do not require authentication so you can specify a user here who has the necessary privileges to receive and send messages to the JMS destinations that are exposed through Diffusion.
<reconnect>	If a connection cannot be made between Diffusion and the JMS server or the connection is severed, the `<max-reconnections>` and `<interval>`

	parameters enable you to specify how many times to retry the connection before giving up and how long to wait between each attempt. A value for `<max-reconnections>` of -1 indicates that the adapter keeps trying to connect forever.
<root-topic>	To provide partitioning of the topic tree between topics related to JMS and other topics, it is necessary for a root topic name is defined here. In the event of more than one JMS adapter, you can segment the topic tree further.
<priority>	JMS supports messages with up to 10 priority levels (0-9) with 0-4 considered to be different grades of normal priority and 5-9 to be different grades of high priority. Diffusion only has the concept of low, medium and high priority. Using this parameter, you can map JMS messages within a given priority range to a representative Diffusion priority, and in the other direction.
<queue-distribution-mode>	Unlike topics, a message on a JMS queue is delivered to only one client. When Diffusion receives a message from a queue, it uses this parameter to determine which of its connected clients subscribed to the corresponding Diffusion topic is selected to receive that message. Valid values are: **SMALLEST_QUEUE** Choose a client with the fewest number of messages outstanding in its message queue from Diffusion. **RANDOM** Select a client randomly.
<artefact-names>	Not all characters in a Diffusion topic name are valid JMS topic or queue names (and the other way around). The two attributes on this element (`jms` and `diffusion`) are lists of characters where the n^{th} character in one is replaced by the corresponding n^{th} character in the other.

JMS Adapter examples

The following scenarios assume that the JMS adapter is configured and connected to a JMS server with a `root-topic` of `jms`.

Receiving messages from JMS

A user wants to receive messages from JMS topic `XYZ`.

Procedure

1. Diffusion client creates a subscription to the topic `jms/topic/XYZ`.
2. Once a message has been sent from the source system into the JMS server, the Diffusion client receives an initial topic load message.
3. Subsequent messages from the source system result in delta messages being delivered to the Diffusion client.

   The same technique is used for receiving messages from JMS queues, with the following differences:

   - Other clients subscribing to the same JMS queue (either through Diffusion or directly using JMS) might receive the message instead of our client.
   - All messages originating from JMS queues are initial topic load messages. Since a sequence of messages from a JMS queue are unlikely to always be delivered to the same client, the concept of delta messages does not readily apply and the full message state must be supplied every time.

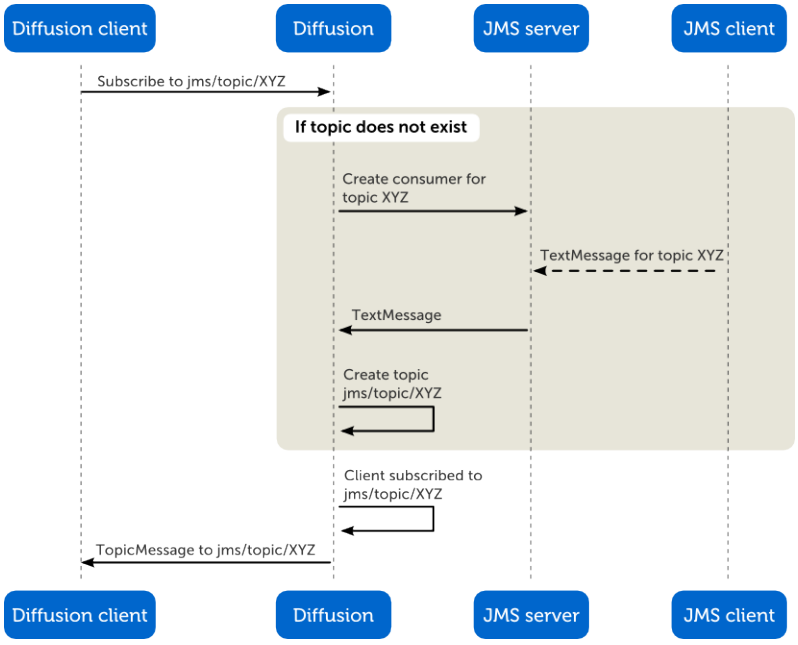


Figure 57: Subscription flow

Sending messages to JMS

A user wants to send messages to the JMS topic `XYZ`.

Procedure

1. The Diffusion client sends a message to the topic `jms/topic/XYZ`.
2. The JMS server receives an equivalent TextMessage.

   Unlike most Diffusion solutions, it is not necessary to subscribe to a Diffusion topic before sending it a message which targets a JMS destination.

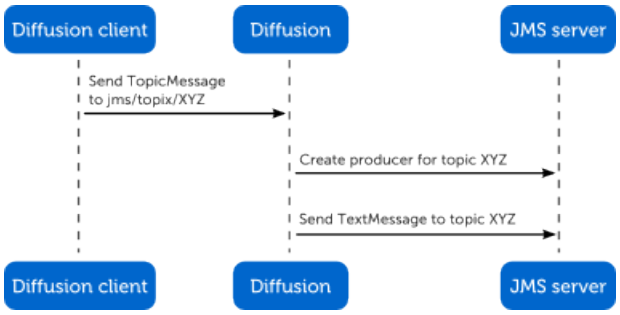


Figure 58: Sending flow from a Diffusion client to a JMS topic (or queue)

Processing a request-reply message with a Diffusion client

A common pattern among JMS solutions is for a client to receive a message from JMS and a reply is expected to be sent to a specific JMS topic or queue. Typically, the JMS publisher creates and sends a message with the `JMSReplyTo` header set to some other destination defined within the JMS server. This can be a topic, queue, or a temporary topic or temporary queue. The Diffusion client does not have to know the destination type as this is handled within the adapter.

Procedure

1. Diffusion client subscribes to topic `jms/queue/ABC`.
2. JMS producer creates a temporary queue (`XYZ`) and subscribes to it.
3. JMS producer sends a message to the queue `ABC` with `JMSReplyTo` set to the queue `XYZ`.
4. Diffusion client receives a message on queue `jms/topic/ABC`, with `DiffusionReplyTo` set to `jms/reply/XYZ`.
5. Diffusion client sends a response message to the queue `jms/reply/XYZ`.
6. JMS producer receives a `TextMessage` on the temporary queue `XYZ`.

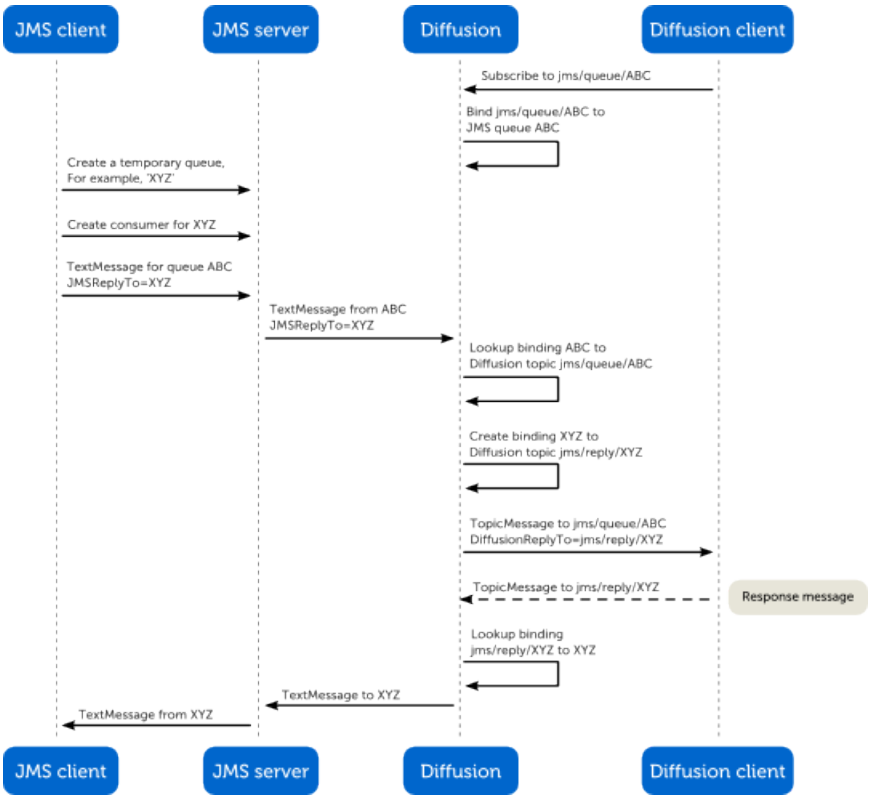


Figure 59: Request-reply initiated by a JMS client and serviced by a Diffusion client

Sending a request-reply message from a Diffusion client

You can send a message from a Diffusion client into a JMS server with the expectation that a JMS client processes the message and send a response back to the same Diffusion client.

Procedure

1. JMS client subscribes to messages on queue `ABC`.
2. Diffusion client subscribes to `jms/tmp/queue/XYZ`. (Commonly, `XYZ` is a unique identifier).
3. Diffusion client sends a request message to `jms/queue/ABC` with the `DiffusionReplyTo` header set with the value `jms/tmp/queue/XYZ`.
4. JMS client receives the request message on queue `ABC`, with the `JMSReplyTo` header set to queue `DEF`.
5. JMS client sends a reply to queue `DEF`.
6. Diffusion client receives the reply on topic `jms/tmp/queue/XYZ`.

> **Note:** The return Diffusion topic can be any topic, so it is not necessary that the originating Diffusion client receives the reply – it can be any client listening for messages on that topic.

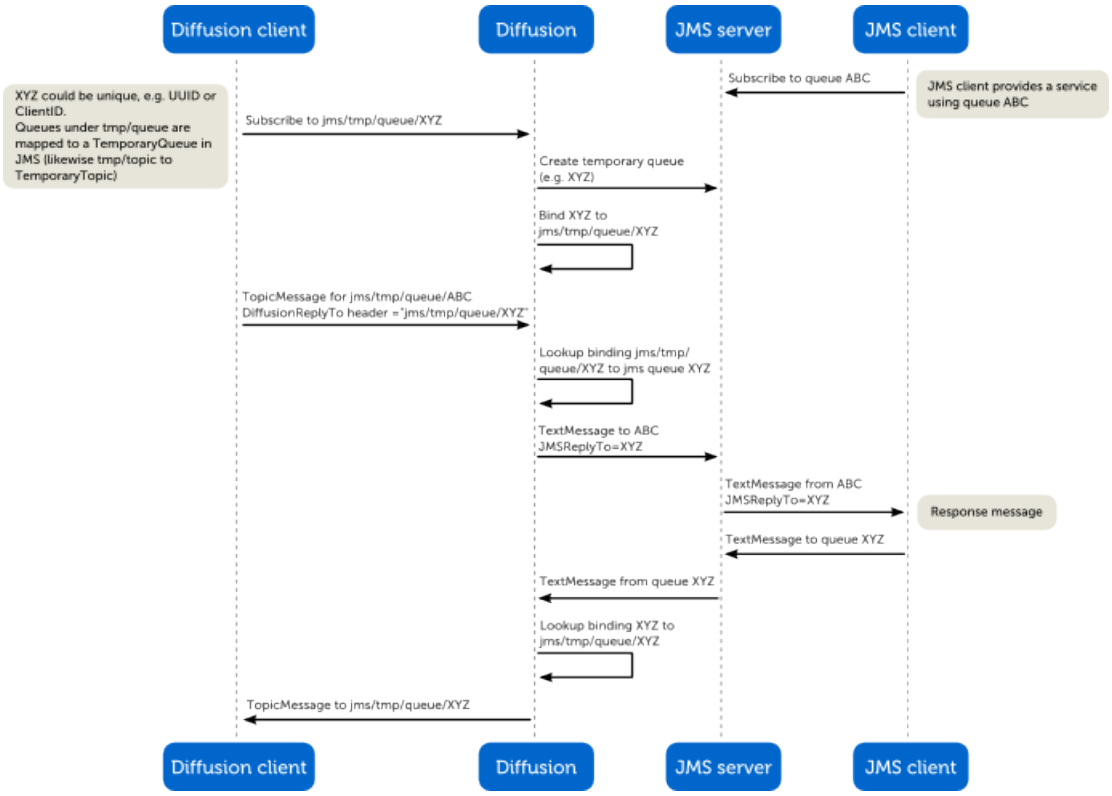

Diffusion client
Diffusion
JMS server
JMS client
XYZ could be unique, e.g. UUID or ClientID.
Queues under tmp/queue are mapped to a TemporaryQueue in JMS (likewise tmp/topic to TemporaryTopic)
Subscribe to queue ABC
JMS client provides a service using queue ABC
Subscribe to jms/tmp/queue/XYZ
Create temporary queue (e.g. XYZ)
Bind XYZ to jms/tmp/queue/XYZ
TopicMessage for jms/tmp/queue/ABC DiffusionReplyTo header ="jms/tmp/queue/XYZ"
Lookup binding jms/tmp/queue/XYZ to jms queue XYZ
TextMessage to ABC JMSReplyTo=XYZ
TextMessage from ABC JMSReplyTo=XYZ
Response message
TextMessage to queue XYZ
TextMessage from queue XYZ
Lookup binding XYZ to jms/tmp/queue/XYZ
TopicMessage to jms/tmp/queue/XYZ
Diffusion client
Diffusion
JMS server
JMS client

Figure 60: Request-reply initiated by a Diffusion client and serviced by a JMS client

Chapter 18

Tuning

In this section:

- Buffer sizing
- Message sizing
- Client queues
- Client multiplexers
- Write selectors
- Connectors
- Thread pools
- Client reconnection
- Client failover
- Client throttling
- Memory considerations
- Platform-specific issues
- Publisher design

Aspects of tuning Diffusion for better performance or resilience

This section covers aspects of configuring Diffusion to achieve higher levels of performance and covers some of the more advanced features which enable users to get more out of Diffusion.

Buffer sizing

There are a number of places within the configuration of Diffusion where buffer sizes must be specified and getting these right can have a significant impact upon performance.

Connector output buffers

An output buffer size must be configured for each connector.

The output buffer size configured for a connector must be at least as large as the largest message that is expected to be sent to any client connecting through that connector. However, the buffer size can be much larger so that the messages can be batched at the server, which improves performance (see below).

Each connected client is assigned a socket buffer of the specified size if possible. A warning is logged if a smaller socket buffer was allocated than requested.

In addition each client multiplexer has a buffer of the configured size (as a multiplexer writes to only one of its clients at any one time). The multiplexer buffer is used to batch messages from the client queue before writing and, if the socket buffer does end up being smaller than the configured buffer and the throughput is high, the allocated socket buffer size might become a bottleneck.

Getting the correct output buffer size is vital. Make this too small and the Diffusion server does not batch and write messages to clients at optimal rates. Make them too big, extra memory is consumed or messages might time out and cause the client connection to be closed.

If the output buffer size is larger than the TCP output buffer size, this can cause problems if the client is slow consuming. If a slow-consuming client does not clear messages from the TCP buffer fast enough, messages on the connector buffer which are waiting to move to the TCP output buffer can time out. You can avoid this problem by setting the TCP output buffers for your operating system and the connector output buffers for your Diffusion server to the same value. You can also increase your message timeout interval.

> **Note:** For maximum performance, ideally all clients configure their input buffer size to match the connector's output buffer size.

Client output buffers

As at the server, the output buffer sizes in use must be configured for a client.

In the Java client this is specified in the ServerDetails object used to make the connection. As the Java client does not buffer messages, this only has to be large enough to cater for the largest message size that is sent to the server.

Publisher client output buffers

A publisher client (a connection made from a publisher to another Diffusion server) is slightly different from a normal client in that it does queue and buffer messages for sending. It is advantageous to throughput to use a larger output buffer size.

The output buffer size is configured in the server element in `Publishers.xml` or in the ServerDetails object depending upon how the connection is being made.

Connector input buffers

Each connector also specifies an input buffer size.

Each client that connects is assigned a buffer of this size to receive inbound messages. This buffer must be as large as the largest message expected.

This size is also used to allocate a receive socket buffer for the client. The socket buffer allocated might actually be less than requested in which case a warning is logged.

For maximum performance, the size used for this buffer must match up with the output buffer size used by clients.

Client input buffers

Clients must also specify the buffer size to use for input.

In the Java client this is specified in the ServerDetails object used to make the connection (or possibly `Publishers.xml` for a publisher client connection).

Matching buffer sizes

For optimal throughput it is desirable to match the size of buffers at each end of every connection. The input buffer size used by clients ideally matches the output buffer size at the connector that they connect to. Also the output buffer size specified by clients must match the input buffer size of the connector they connect to.

> **Note:** Because publisher server connections queue and batch messages at both ends, use a separate connector for such connections such that optimal buffer sizing can be achieved.

Message batching

The size of output buffers configured for a connector can be much larger than the largest expected message size because the server uses these buffers to batch client messages which improves performance. Ideally the buffer size is a multiple of the average message size.

> **Note:** When using for HTTP clients, allow between 20 (for HTTP Duplex) and 250 (for browser) bytes extra for control information.

Each client multiplexer assigns an output buffer of each size specified by client connectors. So if there were 3 client connectors, each specifying a different output buffer size, and 2 client multiplexers, each multiplexer assigns 3 different buffers (6 in total). When using HTTP Duplex connections, each multiplexer assigns an additional buffer for chunked encoding.

When a client multiplexer is unable to write the contents of an output buffer to a client in one go, the writing is deferred and the multiplexer takes a copy of the remaining data in the output buffer into its own temporary buffer.

Message sizing

The sizing of messages that are sent to clients is very important to the overall performance and this must be carefully considered within the design of publishers.

Every topic message has a fixed header of 6, 4 or 3 bytes (depending upon the value of Message Length Size). It then has the topic name terminated by one byte, plus any user header information that is also included with the message.

It is important to work out the size of the message so that the connector buffers can be set correctly, otherwise Diffusion is unable to put the messages on the wire quickly enough.

Byte pinching

With any messaging system, the smaller the messages, the lower the latency and the faster the system performs. There is a consultancy exercise that Push Technology performs as a service to analyze the messages and reduce them as much as possible. The folloinf list includes a few of the best practices to use:

- Only send data that is required by the client.
- Look at the data format and strip any fat off the message.
- Is the information being sent a true delta?
- Client side data models

Message encoding

If you are sending large messages, it is worth compressing the messages. This happens only once on the server, and then the clients have the technology to decompress them, this also includes JavaScript clients. If other encoding is used, it is worth bearing in mind the CPU overhead required.

Client queues

A maximum queue depth can be configured for client queues so that clients are closed if their message backlog becomes too large.

The maximum queue depth must be chosen carefully as a large size might lead to excessive memory usage and vulnerability to Denial of Service attacks, whilst a small size can lead to slow clients being disconnected too frequently.

Client queues do not take any memory, as Diffusion uses a Zero Copy paradigm, but there are consequences in setting them too small or too large. If the client queue is set too small, once the client has filled its queue the Diffusion server closes the client.

When considering queue depth take into account the average message size and publication rate. Messages that are held in the client queue are not garbage collected and can get promoted, which increases their impact on GC pressure. If messages in the client queue build up, consider the maximum delay in the context of you application. For example: Assuming 100 bytes is the average message size and the application is publishing an average of 100 messages per second. If the client queue is setup to have a maximum depth of 1000 messages this means we allow messages to build up for a slow client for up to 10 seconds, during this time a slow client is building up a cache of 100k of messages to be sent.

> **Note:** It is natural for queues to build up a little with spikes in publication rate or momentary bandwidth limits, but the tolerance to such delays is expressed in the client queue depth and must be considered in that context.

Client multiplexers

Tuning multiplexers for optimal performance

The load of batching, conflating and merging messages being sent from publishers to outbound clients is spread across client multiplexers. The number of configured client multiplexers must take into account the expected message load and concurrent client connections. The more clients are assigned to a multiplexer the more load it must contend with.

A client multiplexer processes all client messages into the client queue. Clients are added to the multiplexers according to a load balancing policy (default is round robin, but least clients is also supported).

Publishers either broadcast on a topic to all subscribed clients or send clients direct messages. When broadcasting all multiplexers are notified and go on to find all subscribed clients which are assigned to the particular multiplexer. When a message is sent to a particular client only that client's multiplexer is notified. It is more efficient to broadcast than it is to send the same message to a large number of clients by iterating over them.

Client multiplexers are non-blocking, high priority threads so having too many can be detrimental, as they are competing for the same resource (CPU). As a rule of thumb, the number of multiplexers must not exceed the number of available logical cores. If a client multiplexer becomes over-subscribed, message latency can increase. For maximum throughput, the number of multiplexers can be set to the number of available cores, but this configuration is only recommended in the case where other threads are assumed to be mostly idle (for example, little inbound traffic, low publisher overhead).

Client multiplexers performance is influenced by the use of merge and conflation policies as those are executed in the multiplexer thread. It is recommended that conflation policy changes and in particular changes to merge conflation logic be profiled and written with performance in mind. In particular the use of locks or any other blocking code is highly discouraged.

Each multiplexer uses a different buffer for each output buffer size that is specified to any connector. If there were three connectors with different output buffer sizes specified, each multiplexer assigns three different buffers. Each multiplexer might also assign an extra buffer for HTTP use. A larger output buffer enables more efficient batching of messages per write, as large writes are generally more efficient but care must be taken to not overwhelm client connections regularly and causing them to be blocked for any period of time.

When a multiplexer is unable to write a message to a client because the buffer has become full, a write selector is notified. The write selector is responsible for watching the client and notifying the multiplexer when it becomes writable. The multiplexer remains responsible for writing the message.

Write selectors

Write selectors are used to finish writing partially written buffers being sent to clients by the multiplexer.

Each connected client creates a message channel with the Diffusion server. Each message channel has a socket buffer of configured size. The socket buffer fills as the server sends messages to the client and it empties as messages are written onto the wire. Since Diffusion uses a non-blocking IO model when the socket buffer is full, processing continues if not all of the messages can be written to a single channel. This allows the multiplexer to send messages to other clients. The write selectors handle any remaining bytes that must be sent to the client.

Each write selector is a separate thread that is notified when a partial write happens. When a selector is notified of a partial write the write selector, monitors the message channel and when it becomes writable again it, notifies the multiplexer that it can write more bytes. The multiplexer writes as many of the bytes as it is able to. Write selectors are non-blocking threads so having too many might be detrimental.

The write selectors are configured together as a pool, different write selectors cannot use different configuration values.

Refer to the XML Configuration for information about configuring from the XML files.

Refer to the WriteSelectorConfig API documentation for information on configuring programmatically.

Normally the configured values are sufficient but when a large number of slow consuming clients (for example, SSL) are expected, it might be necessary to increase the size of the pool. The write selectors are used only to notify the multiplexers about slow clients, the multiplexers are more important to performance.

Connectors

You can tune your connectors to handle multiple connections and improve performance.

Configuring multiple connectors

When there is more than one publisher application running on a server, configure a separate connector for each publisher so that buffer requirements can be specific to the connector.

It might also be beneficial to configure different connectors for different client types as their requirements can be different. This is especially true for publisher clients where there are low numbers of connections which benefit from very large buffer sizes in both directions.

Buffers

As Diffusion can have tens of thousands of connections at any one time on a machine it is important to make sure that the buffers are set correctly.

To reduce the memory footprint, the Diffusion server will condition the input and output buffers. If the buffers are set too small, Diffusion cannot write the messages in one go and delegate the task to a writer selector.

There is a separate section devoted to tuning buffers.

Acceptors

The acceptor has the job of accepting the open connection from the client. Having too few of these means that the socket server sustains a backlog and the client will take time to connect or even timeout. Each acceptor is a thread, so having too many drains resources from the system.

Thread pools

Thread pools are used within Diffusion to optimize the use of threads.

It is important to understand balance when tuning thread usage for a system. There must be sufficient thread resources required but not so many as to starve other parts of the system. At the end of the day there are only so many threads that a system can provide.

In general, when provisioning threads, separate the blocking and non-blocking activities. While it is beneficial to have more threads than cores for blocking tasks it is detrimental to the server if more threads than cores are runnable at any given time.

There are a number of places where thread pools are used within Diffusion. These are discussed in the server concurrency section.

Configurable properties

The following key values can be configured for a thread pool to influence its behavior and use of resource:

Table 88: Values that can be configured for a thread pool

Property	Usage
Core size	The core number of threads to have running in the thread pool. Whenever a thread is required a new one is created until this number is reached, even if there are idle threads already in the pool. After reaching this number of threads then at least this number of threads is maintained within the pool.
Maximum size	The maximum number of threads that can be created in the thread pool before tasks are queued. If this is specified as 0, the pool is unbounded and so the task queue size value is ignored. Generally an unbounded pool is not recommended as it can potentially consume all machine resources.

Property	Usage
Queue size	The pool queue size. When the maximum pool size is reached then tasks are queued. If the value is zero, the queue is unbounded. If not zero then the value must be at least 10 (it is automatically adjusted if it is not).
Keep-alive time	The time limit for which threads can remain idle before being terminated. If there are more than the core number of threads currently in the pool, after waiting this amount of time without processing a task, excess threads are terminated. A value of zero (the default) causes excess threads to terminate immediately after executing tasks.
Notification handler	A thread pool can have a notification handler associated with it to handle certain events relating to the pool. This allows for user written actions to be performed (for example, sending an email) when certain pool events (like too much task queuing) occur. See below for more details.
Rejection handler	A thread pool can have a rejection handler associated with it to handle a runnable task that has been rejected. This allows user written actions to handle a runnable task that can not be executed by the thread pool. See below for more details.

Notification handler

A thread pool notification handler can be configured to act upon certain thread pool events.

These events are:

Table 89: Events that a thread pool notification handler can act on

Event	Description
Upper threshold reached	A specified upper threshold for the pool has been reached. This means the pool size has reached the specified size. The event is notified only once and is not notified again until the lower threshold reached event has occurred.
Lower threshold reached	A specified lower threshold for the pool has been reached after an upper threshold reached event has been notified. This means the pool size has now shrunk the specified size.
Task rejected	The pool has rejected a task because there are no idle threads available and the task queue has filled. What happens to the rejected task depends upon the type of pool. Typically, the

Event	Description
	task is run within the thread that passes the task to the pool, which is not desirable. This is why the thread ought to be notified when it occurs. This differs from the rejection handler in that it does not expose the runnable task. This means it can be used only for notification.

The notification handler is a user written class which must implement the `ThreadPoolNotificationHandler` interface in the threads Java API. The name of such a class can be configured for in-bound or out-bound thread pools or for connector thread pools in which case an instance of the class is created (and must have a no arguments constructor) when the thread pool is created.

Rejection handler

A thread pool can have a rejection handler associated with it to handle a runnable task that has been rejected.

Two rejection handlers are provided with Diffusion. These are the `ThreadService.CallerRunsRejectionPolicy` and `ThreadService.AbortRejectionPolicy`.

The `ThreadService.CallerRunsRejectionPolicy` executes the runnable task in the thread that tried to pass the runnable task to the thread service. The `ThreadService.CallerRunsRejectionPolicy` does not execute the task and instead generates an exception.

Note: The inbound thread pool has a mandatory caller runs rejection policy.

The rejection handler is a user written class which must implement the `ThreadPoolRejectionHandler` interface in the threads Java API. The name of such a class can be configured for in-bound or out-bound thread pools or for connector thread pools in which case an instance of the class is created (and must have a no arguments constructor) when the thread pool is created.

Adjusting the configuration

Adjust thread pools gradually. Ideally, duplicate expected maximum loads in test environment. This environment can be used to tune the thread pools to satisfy the load. Tune the thread pools so they are just able to cope with the maximum load, increasing them beyond this might degrade overall performance.

Background thread pool:

In general, the defaults suffice for the tasks assigned to the background thread pool by Diffusion. If you assign tasks to the pool yourself, consider increasing the number of threads.

Inbound thread pool:

This pool is used to handle inbound connections and messages. Increasing the thread pool allows new connections and received messages to be handled over a greater number of threads. However, much of the behavior in this pool can involve locking the clients or parts of the topic tree. This can cause lock contention that delays processing.

Due to the underlying implementation of Java NIO sockets a high rate of threads being added/removed from the incoming thread pool will result in the allocation of off-heap byte buffers. In extreme cases this can result in an out of memory exception being thrown as the server runs out of off heap allocation space.

Client reconnection

You can configure the client reconnection feature by configuring the connectors at the Diffusion server to keep alive the client session.

Normally when a client application loses its connection to the server, perhaps due to some communications error, the only option is for it to connect again and then re-establish the state of the topics to which it was subscribed. There is however, the facility for clients to be able to reconnect a lost connection without losing its topic state or messages that were queued for it whilst it was not connected.

This can be useful for mobile clients where connections are less reliable.

Server configuration

To enable clients to reconnect, connectors must be configured to keep alive client sessions that are in the DISCONNECTED state. To do this a keep-alive time must be specified for the connector.

If a client signals that it wants to disconnect or stop, the client state on the server is removed when the client disconnects or stops. However, in all other circumstances where the client loses connection, the client goes into the DISCONNECTED state, where the subscriptions are retained and messages are queued as normal for the amount of time specified by the keep-alive time of the connector.

If the server is configured to expect reconnecting clients, clients that are not currently connected and might reconnect are excluded from the regular system pings that the server sends to clients. The reason for this behavior is that the keep-alive time might be longer than the interval between system pings. If this is the case, the reconnecting client is unable to respond to the pings from the server. Hence, no pings are sent from the server to a reconnecting client.

If the client then reconnects before the keep alive period has expired, the sending of messages to the client resumes from the point when the failure occurred. Messages that were in transit at the time of disconnection might be lost.

The only way to guarantee the delivery of messages on reconnection is for the publisher to mark the messages as requiring acknowledgment as any such messages that have been sent but not acknowledged on reconnection are requeued for the client. The delivery of acked messages is guaranteed. However, because an ack from the client might have been lost during the disconnection, there is the possibility that a message might be delivered to the client twice in this scenario.

It is important when using acknowledged messages to guarantee delivery after reconnect that the ack timeout set for messages is sufficiently long to allow for the time that a message can be queued for a client plus the keep alive time configured for reconnection.

If an ack timeout expires before a message is even dequeued for a client, the non-acknowledgment is notified and the message is not sent to the client.

Message queue management

Managing message queues when using client reconnection

When a client session is being kept alive, messages for the client continue queuing for the client until the keep alive period expires or the client reconnects. This puts an unusual load on the client queue and the facility exists to adjust the maximum client queue depth for the period of disconnection.

This is done by specifying a keep alive queue depth which is greater than the normal maximum queue depth. When disconnected the queue can expand to the higher value and when reconnection occurs and the queue starts to drain, when the queue size goes down to a value of 80% of its previous limit, the maximum queue depth reverts to the normal value.

The keep alive queue depth has an effect only if it represents a value higher than the normal maximum queue depth.

Client reconnection

Configuring the reconnection of clients

Not all clients support reconnection but those that do have a reconnect method which they can call on notification of a lost connection.

If the reconnection succeeds, the client is subscribed to all of the same topics as before and starts to receive messages again, including all of those queued whilst the client was disconnected.

Messages in transit at the time of disconnection might be lost, however any message marked as requiring acknowledgment and sent by the server that was not acknowledged by the client is retransmitted on reconnect. The delivery of acked messages from client to server is guaranteed on reconnect although there is the possibility that the client might receive a message it had acked before the connection again after reconnection if the ack had never reached the server.

A reconnection might not succeed, either because keep alive is not specified on the connector that the client has connected to, or the keep alive time period has expired. In this case a normal new connection is established with the same topic set as was specified on the original connection.

How to test reconnection in my environment?

To simulate a communication error, we use a proxy between the client and the server.

- Start Diffusion. (By default it uses port 8080)
- Set the proxy to listen on a different port (for example, 9090) and redirect the connection to 8080
- Connect the client through the proxy on port 9090.
- Kill the proxy.
- Start the proxy.
- Reconnect the client.

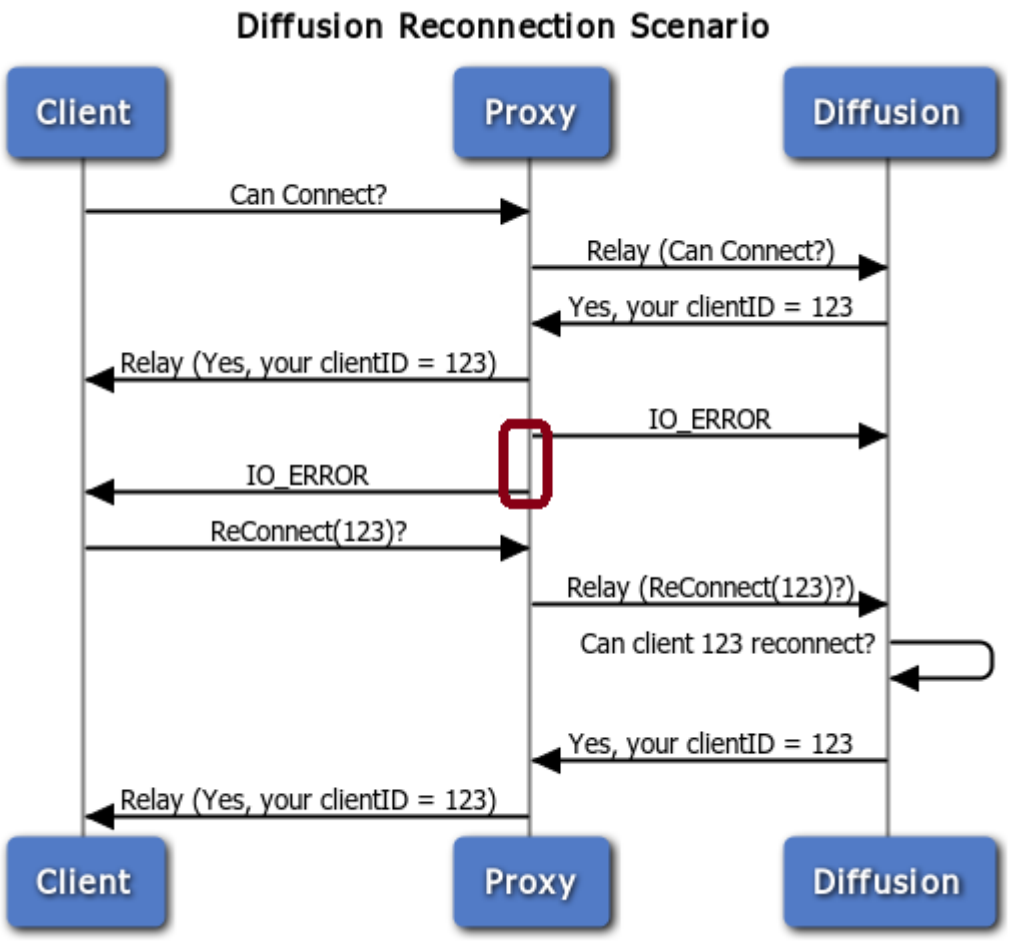


Figure 61: Reconnection scenario

Note: The proxy behaviors are different depending on the operating system and the TCP/IP stack configuration.

Common errors

1. From the client, request a connection close and call reconnect.

   After a close request, the client cannot reconnect. In this case, the client will establish a new connection with a different client ID assigned by Diffusion.
2. Unplug the network wire from the computer where the client is running

   This will not throw an IO_ERROR in the other end of the connection.

Client failover

You can configure a client to failover to another Diffusion server after it loses connection to the Diffusion server it was previously connected to.

Client failover is when a client loses its connection to a server and attempts to connect to a different one. The client is provided with a list of servers. If a client loses its connection to a server it can automatically attempt to connect to the next server in the list. If it fails to connect or loses its connection to that server, it tries the next server on the list. This is referred to as autofailover. Generally the list of servers to connect to must be provided before attempting to make the connection. How the list of servers is provided differs between client APIs and the JavaScript client does not support autofailover but it can be implemented using the callback methods.

Using automatic failover

If a client has an established connection that it loses, autofailover attempts to open a new connection in the next connection in the list. This is not compatible with reconnection because reconnection attempts to preserve the state of the client (the client ID and the subscribed topics). As the new server has no knowledge of the client it is unable to preserve this state. Autofailover must be enabled and a list of servers to connect to provided.

Using load balancing with autofailover

You can enable load balancing in conjunction with autofailover. When load balancing is enabled and a client loses connection, the list of servers is shuffled before the client selects the next server to attempt to connect to.

In Java, for example, you can enable load balancing by using the `setLoadBalancing` method on the `ConnectionDetails` object.

Using server cascading

When a client attempts to place a connection, if the attempt fails, the next server in the list is chosen. Server cascading is similar to autofailover except this logic is applied prior to a connection, whereas autofailover applies once a connection is in place.

In Java, for example, you can enable server cascading by using the `setCascading` method on the `ConnectionDetails` object.

> **Note:** Server cascading is different to protocol cascading, which attempts to connect to the same server using different protocols before a connection has been opened.

Java failover

Configuring failover in Java

In Java the ConnectionDetails object supports a collection of ServerDetails objects. The server details are use the control failover between servers. The ConnectionDetails factory methods provide several options for creating ConnectionDetails with multiple ServerDetails objects. A collection of ServerDetails object can be passed as a parameter, a varargs method supports ServerDetails and another varargs method supports String URLs, which ServerDetails objects are constructed from. After construction the ServerDetails objects used by the ConnectionDetails can be altered by calling the `setServerDetails(Collection<ServerDetails>)` method.

The following code supports autofailover from the server with the IP address 192.168.0.1 to the server 192.168.0.2. If the client loses connection to 192.168.0.1 it tries to connect to 192.168.0.2. It uses the varargs method to create a ConnectionDetails object with multiple ServerDetails objects constructed from String URLs.

```
ConnectionDetails details =
    ConnectionFactory.createConnectionDetails(
        "dpt://192.168.0.1:8080",
        "dpt://192.168.0.2:8080");
details.setAutoFailover(true);
ExternalClientConnection client = new ExternalClientConnection(listener,
 details);
client.connect();
```

For further information refer to the Java API documentation for ConnectionDetails and ServerDetails.

JavaScript failover

Using failover in JavaScript

The JavaScript client does not support autofailover. Support for failover is limited. If the connection attempt fails DiffusionClient.connect(DiffusionClientConnectionDetails) can be called with a different object. You must provide the logic to do this on connection failure.

ActionScript failover

Using failover in ActionScript

In ActionScript the ConnectionDetails object supports an array of ServerDetails objects. The server details are use the control failover between servers. The ConnectionDetails constructor has a mandatory ServerDetails object. After construction additional ServerDetails objects used by the ConnectionDetails can be altered by calling the `addServerDetails(ServerDetails)` method and the `setServerDetailsArray(Array)` method.

This code:

```
var server0:ServerDetails = new ServerDetails("dpt://192.168.0.1:8080");
var server1:ServerDetails = new ServerDetails("dpt://192.168.0.2:8080");
var details:ConnectionDetails = new ConnectionDetails(server0);
details.addServerDetails(server1);
details.setAutoFailover(true);
var client:DiffusionClient = new DiffusionClient();
client.connect(details);
```

supports autofailover from the server with the IP address 192.168.0.1 to the server 192.168.0.2. If the client loses connection to 192.168.0.1, it tries to connect to 192.168.0.2. It uses the `addServerDetails(ServerDetails)` method to add a single additional server to connect to. For further information refer to the Flex Classic API documentation for ConnectionDetails and ServerDetails.

Client throttling

Throttling is a method of ensuring that the Diffusion server limits (throttle) the volume of messages it transmits to a client within a specified period of time. This can be used to limit bandwidth usage or to prevent more messages being sent to a client than it can cope with.

How does throttling work?

Throttling is applied to a client queue so that the number or volume of messages sent to that client is restricted. Diffusion only dequeues a message to send to a client if that client has not breached its throttling limit.

Throttle types;

- Messages per second (Only a specified number of messages are sent every second.)
- Bytes per second (Only a specified number of bytes are sent every second)
- Message interval (A single message is sent every n milliseconds.)
- Buffer interval (A full output buffer (or the equivalent of) is sent every n milliseconds)

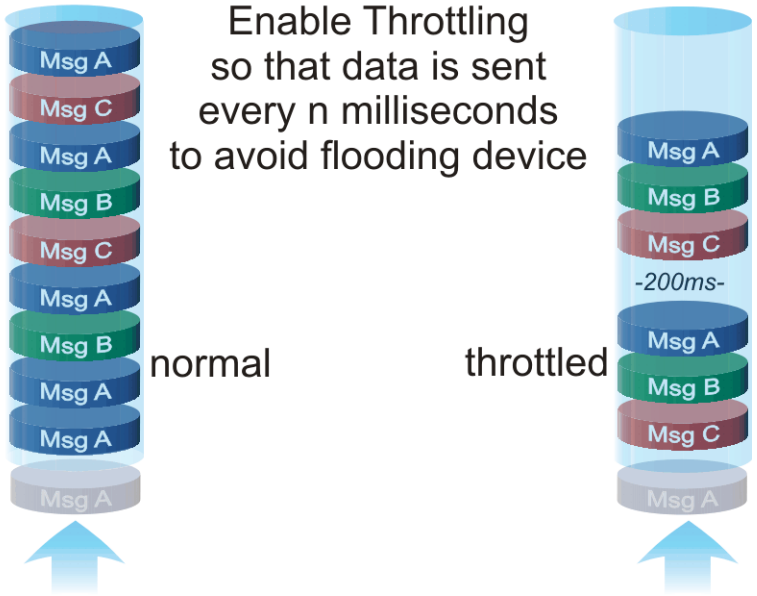


Figure 62: Normal and throttled client queues

Enabling throttling

Throttling can be enabled on a client-by-client basis from within the publisher.

To throttle a `Client.throttle` method which allows you to specify the type of throttling and a limit. A `clientThrottler` reference is returned.

If the throttle method is called for a client that is already throttled, it has the effect of removing the old throttler and adding a new one.

Call the `Client.removeThrottler` method to stop throttling.

The `Client.isThrottling` method can be used to determine whether a client is currently throttled and, if it is, the getThrottler method can be used to determine the type of throttling and the current limit.

Memory considerations

Memory usage in the Java VM

Garbage collection (Java HotSpot VM)

Typically you do not have to alter the default garbage collection for the Java VM, but, depending upon the nature of the application, better performance and memory usage might be obtained by tuning the VM's garbage collection parameters.s

Garbage collection tuning is a complex area and consult the Oracle guides to garbage collection tuning. However, some considerations are mentioned below:

Generational vs incremental collection

The default garbage collection algorithm used is generational which means it views newer object references differently from older ones and assumes that what is to be collected is generally in the newer space.

This approach is best for most cases, but in the case where Diffusion is maintaining message queues for clients, messages might remain in a queue for long enough to be moved out of the space eligible for regular collection. The overall effect of this might be that memory (heap) usage appears to grow indefinitely and a sudden peak of memory usage might result in an out of

memory error. This can be overcome by use of the incremental garbage collector which will do period sweeps of older objects. You can do this by using the -Xincgc flag for the Diffusion Java VM.

Platform-specific issues

To run Diffusion it might be necessary to increase the number of sockets and reduce timewait.

It might also be necessary to increase the number of open files that is allowed on UNIX or Linux systems

Socket issues

To fix these problems, complete the following steps based on platform.

Windows

Setting values on Windows

Setting TCP timed wait

This parameter determines the length of time that a connection stays in the TIME_WAIT state when it is closed. When a connection is in the TIME_WAIT state, the socket pair cannot be reused. This is also known as the 2MSL state because the value is twice the maximum segment lifetime on the network. See RFC 793 for further details.

Add TcpTimedWaitDelay registry values as a workaround. You can set these values through REGEDIT command.

Set TcpTimedWaitDelay to 30:

1. Select **Start** > **Run**.
2. In the available field, enter `regedit`.
3. Go to the key directory file: `HKEY_LOCAL_MACHINE/SYSTEM/CurrentControlSet/Services/Tcpip/Parameters/TcpTimedWaitDelay`. The value type is `REG_DWORD`.
4. Double-click **TcpTimedWaitDelay**.
5. Select **Decimal**.
6. Type `30` in the **Value** data field. The default value for this field is 0xF0 (240 decimal). The valid range is 30-300 (decimal).

Setting MaxUserPort

This parameter controls the maximum port number used when an application requests any available user port from the system. Normally, short-lived ports are allocated in the range from 1024 through 5000. Setting this parameter to a value outside of the valid range causes the nearest valid value to be used (5000 or 65534).

Add MaxUserPort registry values as a workaround. You can set these values through REGEDIT command.

Set the MaxUserPort to 65534

1. Select **Start** > **Run**.
2. In the available field, enter `regedit`.
3. Go to the key directory file: `HKEY_LOCAL_MACHINE/SYSTEM/CurrentControlSet/Services/Tcpip/Parameters/MaxUserPort`. The value type is `REG_DWORD`.
4. Double-click **MaxUserPort**.
5. Select **Decimal**.
6. Type `65534` in the **Value** data field. The default value for this field is 0x1388 (5000 decimal). The valid range is 5000 – 65534(decimal).

Linux

Configuring sockets values on Linux

Decrease the time wait before closing the sockets by entering:

```
# echo 3 > /proc/sys/net/ipv4/tcp_fin_timeout
```

Sometimes systems are now configured to prevent one from using a large number of ports, check the port range and modify if required.

```
# cat /proc/sys/net/ipv4/ip_local_port_range
```

This can be increased by issuing the following command

```
# echo "1025 65535" > /proc/sys/net/ipv4/ip_local_port_range
```

To have these new values take effect you might have to do (as root)

```
# /etc/rc.d/init.d/network restart
```

If you want these new values to survive across reboots you can at them to `/etc/sysctl.conf`.

```
# Allowed local port range
net.ipv4.ip_local_port_range = 1025 65535
# net.ipv4.tcp_tw_recycle = 1
net.ipv4.tcp_fin_timeout = 3
```

Increasing number of open files

How to increase the number of open files on UNIX systems.

The `ulimit` command can increase the number of open files. This can be done as a global setting for the box by your network administrator. If this setting cannot be changed globally, the start script for Diffusion must be modified. Edit the `startdiffusion.sh` file and add the following line at the start:

```
ulimit -n 8192
```

The value can be changed to any suitable value. This command can be run only by a user with the right permissions.

Publisher design

Considerations when designing a publisher

Consider the following points when designing and writing a publisher:

Data modeling

The way that the data is fed to a publisher and the way in which the state of the data is maintained within a publisher is key to good performance. Keep message sizes to a minimum and this can be achieved using fine data granularity enabled by hierarchic topics.

Caching

Cache messages wherever possible rather than building new ones every time one must be sent. This particularly applies to topic load messages which can be cached to send to every new client that subscribes, and rebuilt only when the data actually changes. The ideal place to keep such cached messages is with a data object attached to the topic (see topic data pattern).

String handling

Building of Strings by concatenation is very inefficient in Java. Keep String concatenation to a minimum. When String content is used, message caching can help to some degree and wherever possible cache Strings that must be built.

Conditional processing

Excessive use of conditional processing (Checking of topic names, and so on) can be expensive. Use of the topic data pattern can significantly reduce the need for such processing when many topics are in use.

Topic naming

As every message must carry the topic name, long topic names can lead to a large amount of data traffic which can be disproportional to the data being carried. This can be a particular problem when using hierarchical topics. A solution is to use the topic aliasing feature so that only short aliases of the topic names are transmitted in messages.

Concurrency

Concurrent programming means that access to data often must be synchronized but care must be taken not to synchronize more than is necessary as performance can be significantly affected.

Chapter 19

Diagnostics

In this section:

- Logging
- Logging API
- Connection counts
- Message diagnostics
- JavaScript diagnostics
- Flex and Flash diagnostics
- Windows diagnostics
- Debugging a publisher
- Log Messages

Diffusion provides various mechanisms for logging information and diagnosing problems. The major internet browsers also include useful diagnostic tools. All are discussed in this section.

Logging (Java)	Diffusion provides a comprehensive logging facility which can be used from within Java, whether in publishers or applications using Java APIs.
Message diagnostics (Java)	Analyzing message content from within the Java APIs.
JavaScript diagnostics	Analyzing message content and issues utilizing browser tools.
Flex/Flash diagnostics	Analyzing issues within the Flash framework.
Windows diagnostics	Analyzing issues within the .NET APIs.
Debugging a publisher	Being able to step through debug a publisher using the Eclipse IDE

Logging

The Diffusion server logs messages about its activities and any problems that occur. Log messages are written to the console (the process output, which can appear in a terminal window) and to a log file. The level of logging can be adjusted, from a moderate rate of messages under normal circumstances for system monitoring, to very detailed logging for problem diagnosis.

Configuring logging

The log messages created by the Diffusion server, and by publishers deployed to the server, are filtered by the configuration in `etc/Logs.xml`.

You can configure the following aspects of logging:

- The level of logging to the console
- The level of logging to a file
- The name and location of the file
- Whether the log files rotate based on time or file size or both
- The time interval to use to rotate the files
- The file size to use to rotate the files
- The number of old log files to keep

Application code that uses the Diffusion API but is not deployed in a server environment (that is, code that is not part of a publisher) can still use the logging facilities. In this case, the API can be used to reconfigure the logging back-end, and `etc/Logs.xml` configuration does not play a part.

Log levels

The log format was standardized in Diffusion 4.6. The log levels used in the log output changed from a variant of `java.util.logging` values to SLF4J values. To maintain compatibility with existing Diffusion deployments, the `java.util.logging` values are still used in the configuration in this release.

The log levels, in severity order, are as follows:

Table 90: Log levels

Configured value	Messages that are written to logs (SLF4J values)
`SEVERE`	`ERROR` messages only.
`WARNING`	`WARN` and `ERROR` messages.
`ADVICE`	`WARN` and `ERROR` messages. `INFO` messages logged using `LogWriter.advice()`
`INFO`	`INFO`, `WARN`, and `ERROR` messages.
`FINE`	`DEBUG`, `INFO`, `WARN`, and `ERROR` messages.
`FINEST`	`TRACE`, `DEBUG`, `INFO`, `WARN`, and `ERROR` messages.

Warning: Logging can use considerable CPU resources Enable only significant log messages (`INFO` and above) in a production environment. Performance degrades significantly when running at finer logging levels as more messages are produced, each requiring processing.

Log format

From Diffusion 4.6, the log format is formally documented. This eases integration with log monitoring tools.

Each log line is made up of a number of fields. All of the fields except for the Exception are formatted on a single line, delimited by pipe (|) characters.

```
2013-07-17 15:21:57.355|INFO|main|PUSH-000165|Diffusion Server started.|
com.pushtechnology.diffusion.DiffusionController
```

If a log event has an exception, the exception message and stack trace directly follows the message line.

```
2013-07-17 16:57:50.659|WARN|main|PUSH-0000258|
Failed to load publisher 'ConsolePublisher'.|
com.pushtechnology.diffusion.publisher.PublisherManagerInstance
com.pushtechnology.diffusion.DiffusionException: Class
 com.pushtechnology.diffusion.apps.console.publisher.ConsolePublisher for
 Publisher ConsolePublisher not found
 at ...PublisherManagerInstance.createInternalPublisher(Publisher...)
 at ...PublisherManagerInstance.loadPublisher(PublisherManagerInstance.java:108)
 at ...PublisherManagerInstance.loadPublishers(PublisherManagerInstance.java:141)
 at ...PublisherManager.loadPublishers(PublisherManager.java:51)
 at ...DiffusionController.startImpl(DiffusionController.java:220)
 at ...DiffusionController.start(DiffusionController.java:126)
 at ...api.server.DiffusionServer.start(DiffusionServer.java:105)
 at ...api.server.DiffusionServer.main(DiffusionServer.java:140)
 at ...Diffusion.main(Diffusion.java:20)
```

The meaning of each field is described in the following table.

Table 91: Fields included in the logs

Field	Optional or Mandatory	Format/values stable between releases	Description
Time stamp	Mandatory	Yes	The time and date the log event occurred. Asynchronous logging is enabled by default. The server might log a message in a different thread to the one that produced the log event, and at a slightly later time. Consequently, log lines might not be logged in exact time stamp order. The time stamp is displayed using the timezone configured for the Java Virtual Machine (JVM) running the server. The date format can be changed in the `Server.xml` configuration file.

Field	Optional or Mandatory	Format/values stable between releases	Description
Severity	Mandatory	Yes	The log severity, using the SLF4J levels: `ERROR`, `WARN`, `INFO`, `DEBUG`, `TRACE`.
Thread	Mandatory	No	The name of the Java thread that logged the event.
Code	Optional	Yes	A code identifying the message type. All documented messages have a code. Example: `PUSH-000123`. These messages can be logged at any level, but are typically `INFO` or above.
Message	Mandatory	No	A natural language description of the event.
Logger name	Mandatory	No	The logger name. Usually the fully qualified name of the Java class that produced the event.
Exception	Optional	No	If the log event has an associated Java `Throwable`, the exception message and stack trace directly follows the message line.

Optional fields are empty if the log event does not have the information.

The third column indicates whether fields are stable between releases. Where possible, Push Technology will not change the format or values of these fields so they can be relied on for automated log monitoring. The fields not marked as stable are more likely to change between releases, including patch releases.

Log headers

In this release, Diffusion log files start with a special header.

```
2013-07-08 09:58:12.095 : Starting log for Diffusion 4.5.4_01 (Server)
 (2013-06-07T01:02:19)
```

This header shall be replaced with a standard log message in a future release, and is not to be depended on.

Logging API

The Diffusion Java API includes a logging facility that can be used by applications to log information.

Prefer SLF4J

From Diffusion 4.6, the product implementation has switched to the SLF4J logging API.

In this release, the `LogWriter` API described below is still supported for application use. We recommend that new applications use SLF4J to log events instead of `LogWriter`.

Whether SLF4J or `LogWriter` is used, log events are processed by a shared logging back-end and are written to the same destinations.

The LogWriter API (deprecated)

Diffusion provides its own `LogWriter` interface that provides similar functionality to the JDK's `java.util.logging.Logger`. We recommend that new applications use SLF4J to log events instead of `LogWriter`.

Every publisher has its own `LogWriter` which can be obtained using the `Publisher.getLogger()` method. You can use the configuration settings to direct logging performed from within a publisher to the console or files or both.

A `LogWriter` can also be obtained using the `Logs.getLogger()` method. This can be used in non-publisher APIs, and when not running in a server environment. By default, logging outside of the server writes messages to `System.err` and be filtered at `INFO` level. This can be changed by configuring a new handler.

`LogWriter` has a method for each level of logging (`SEVERE`, `WARNING` etc). The method you use determines the log level at which the message is logged. For example, if you use the `fine` method, the message is logged only if `FINE` logging (or lower) is enabled in the configuration.

Within a publisher, logging can be done as follows.

```
protected void publisherStarted() throws APIException {
    getLogger().info("Publisher "+this+" has started");
}
```

In contrast, from within an application class running outside the server environment, logging can be done as follows.

```
Logs.info("A Message To Display");
```

`LogWriter` has methods to check whether a particular level of logging is enabled. This can be used to improve performance and avoid the need to format messages that are never going to be used. For example:

```
LogWriter logger = getLogger();
//..
if (logger.isFineLogging()) {
    logger.fine("Publisher " + this + " has generated some data");
}
```

It is recommended that the above technique is used in all but the highest levels of logging (that is, those that probably are always enabled).

`LogWriter` has versions of the level logging methods which also take a `Throwable` parameter. This allows an associated exception to be logged along with the message. A full stack trace of any exception specified is logged.

Mapping of LogWriter log levels

As described in Configuring logging on page 514, the log levels were changed in Diffusion 4.6. The following table describes the mapping of the `LogWriter` methods to the level used in the log files.

Table 92: Mapping between LogWriter methods and Diffusion log levels

`LogWriter` method	Level used in log file
`severe()`	`ERROR`
`warning()`	`WARN`
`advice()`	`INFO`
`info()`	`INFO`
`fine()`	`DEBUG`
`finest()`	`TRACE`

The logging back-end

SLF4J and `LogWriter` are both APIs that separate the code used to log a message from the logging implementation. The work of formatting and writing out messages is done by the logging back-end.

In this release, the Diffusion logging back-end is based on `java.util.logging`. The back-end implementation of logging might change in a future major release. The information in this section is subject to change.

In a Diffusion server environment there are two log handlers – implementations of `java.util.logging.Handler`. One handler is responsible for writing messages to the console and the other for writing messages a file. In a non-server environment, there is a single default handler that logs to `System.err`.

In either case, additional handlers can be added to process log messages in any way that is required. For example, you might configure a logger that sends messages directly to a UNIX `syslog` process.

A handler can be added using the `LogWriter.setHandler()` method. You can add a handler that processes messages from a single publisher by adding it to the `LogWriter` returned by its `getLogger()` method. Alternatively you can add a handler to the root `LogWriter` obtained from `Logs.getLogger()` and it handles messages for all loggers, including those used by the Diffusion software.

In a server environment, additional handlers do not replace the default handlers. In a non-server environment, the first handler added to the root `LogWriter` replaces the default handler.

Connection counts

The Diffusion server produces connection summaries.

At one minute past midnight Diffusion creates an entry in the file `logs/ConnectionCount`, and resets the counter.

The value in the third column is the total number of client connections that have been made that day.

The value in the fourth column is the maximum number of concurrent connections that have been active that day.

The scheduling of this reset defaults to once per day, but it can be controlled by supplying the `<reset-frequency>` element in `etc/Statistics.xml`.

An example is shown here:

```
2013-03-27 00:01:40     4.4.0_05     128     31
2013-03-28 00:01:48     4.4.0_05     139     28
2013-03-29 00:01:56     4.4.0_05     177     28
2013-03-30 00:01:05     4.4.0_05     118     41
2013-03-31 00:01:13     4.4.0_05     196     30
2013-04-01 00:01:22     4.4.0_05     207     36
2013-04-02 00:01:31     4.4.0_05     188     19
2013-04-03 00:01:41     4.4.0_05     244     44
2013-04-04 00:01:41     4.4.0_05     188     26
2013-04-05 00:01:41     4.4.0_05     195     39
```

Message diagnostics

Within the Java API the `Message.toString()` method always produces a String representation of the message content including headers.

The `Message.toString()` method is very useful for diagnostic purposes. Use it with caution, because there is a significant overhead to converting a message to String format.

A message is rendered in String form as follows:

`(L=` *n* `,T=` *n* `,E=` *n* `)[` *headers* `]` *data*

`L=` *n* shows the message length, in bytes, including headers.

`T=` *n* shows the message type (see protocol for possible values).

`E=` *n* shows the message encoding (see Protocol for possible values).

[*headers*] shows any message headers. The *headers* String shows the headers separated by `<FD>` and terminated by `<RD>` (symbolizing field and record delimiters). Fixed headers appear first followed by any user headers. If the message has no headers, this part is not be present.

data shows a character representation of the data content of the message. `<FD>` and `<RD>` are used to indicate record and field delimiters, if present. If the message is encoded, the decoded data is displayed. If the message has no data content, this is not present.

> **Note:** The message length shown does not necessarily correspond to what is shown. For example, the length of a message with compressed encoding might be significantly less than the length of the data displayed. The part within `()` is always fixed at 6 bytes, the part within `[]` is UTF-8 encoded characters, and the *data* can also be character- or byte-encoded which can affect its length within the message.

The following example of code is followed by the actual message representation that comes from the `Message.toString` method:

```
message.setHeaders("H1","H2");
message.putRecord("A","B","C");
message.putRecord("D","E","F");
System.out.println(message);
```

This example is displayed as:

```
(L=31,T=21,E=0)[MyTopic<FD>H1<FD>H2<RD>]A<FD>B<FD>C<RD>D<FD>E<FD>F
```

> **Note:** There is no field delimiter at the end, because the field delimiter is a separator and not a terminator. A field delimiter at the end implies an empty (zero length) field at the end of the data. The same applies to record delimiters.

JavaScript diagnostics

This section describes how to enable and make use of the browser consoles.

There are currently more than 30 web browsers in use. This section discusses some of the more widely used browsers, namely Firefox, Chrome, Internet Explorer, Opera, and Safari.

As JavaScript is a typeless language, it can very difficult to debug. DiffusionClient surrounds both onMessage and topic listener function invocations with a try..catch block. This might mean that if there is an error in the functions that accept WebClientMessage, this might silently fail and not produce the expected results. To trap this, there is a DiffusionClient.trace(message) function. The trace function creates a message which can be seen in the browser's logging console. Runtime errors are also displayed within the console.

To allow debugging of the client, include the following entry:

```
<script type="text/javascript"> function startDiffusion()
 { DiffusionClient.setDebugging(true); } </script>
```

Mozilla Firefox

Firebug integrates with Firefox allowing you to debug and monitor JavaScript, and edit HTML and CSS. Firebug requires Firefox 3.5 or higher and can be downloaded from the following location: https://addons.mozilla.org/en-US/firefox/addon/1843

To start debugging, click on the Firebug icon in the bottom right corner of the screen.

Ensure that logging is enabled in the drop-down menus for **Console** and **Net**

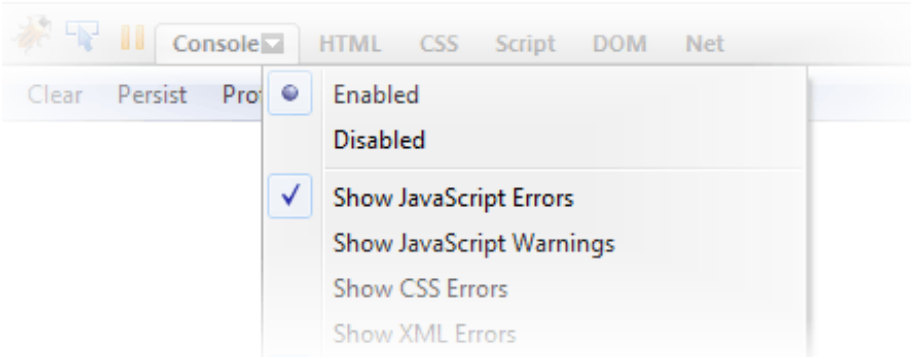


Figure 63: Firefox Console

Google Chrome

Chrome's console can be found in the **Developer** > **Developer tools** section of the menu.

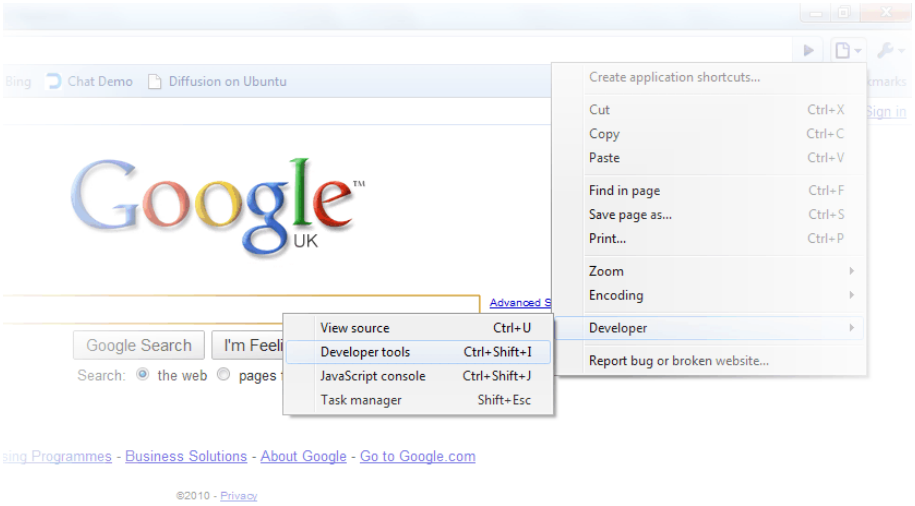
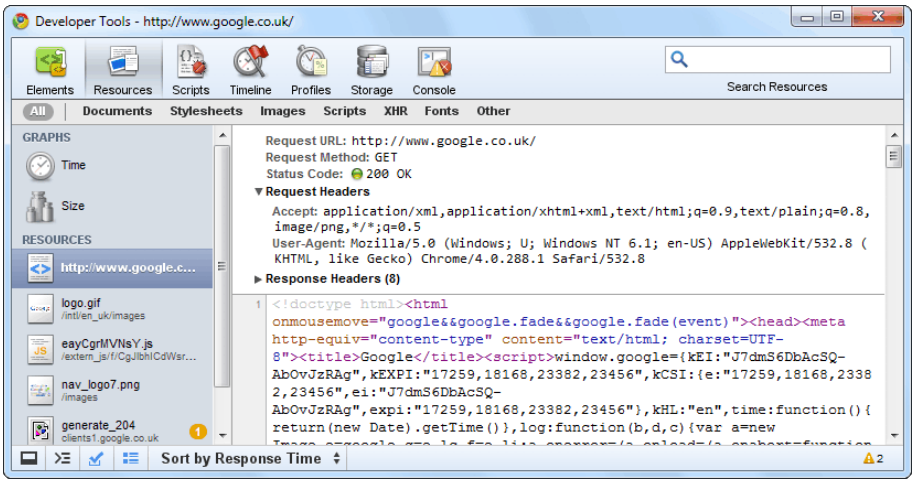


Figure 64: Chrome's console

On the console, go to the **Resource** tab and click **Enable**.



Internet Explorer

IE8 has a console which is accessed by selecting **Tools** > **Developer Tools** of by pressing `F12`.

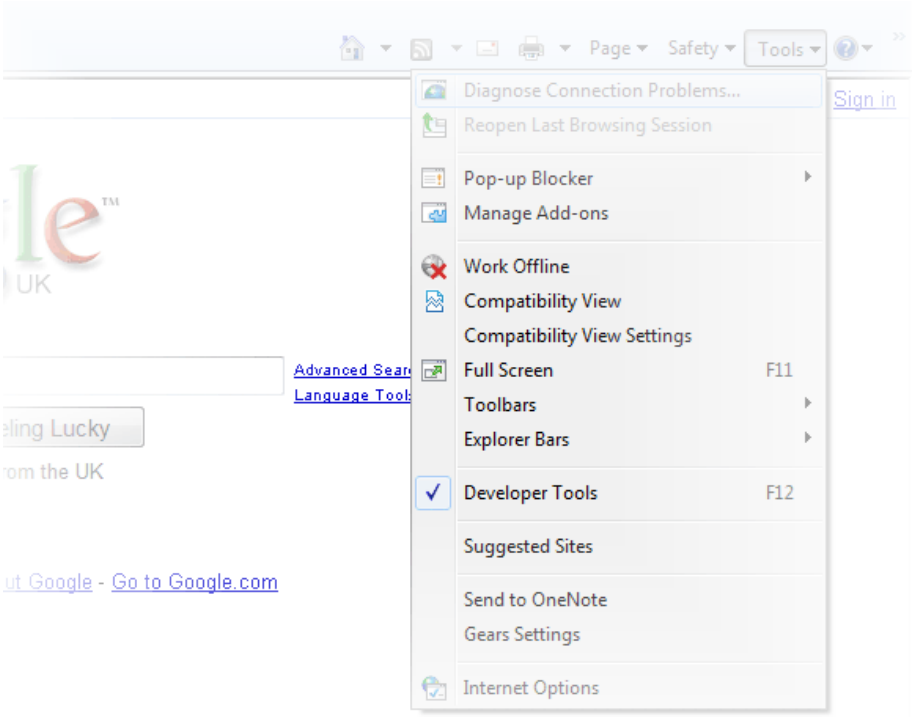


Figure 65: Internet Explorer console

Doing this brings up the console. In some cases when attempting to start Developer tools, the window fails to appear when it is outside the viewport. This can be remedied by pressing `Alt +Space` and choosing **restore**.

Opera

The Opera console is similar to that of Firefox and is accessed through the **Tools** > **Advanced** > **Developer Tools** menus.

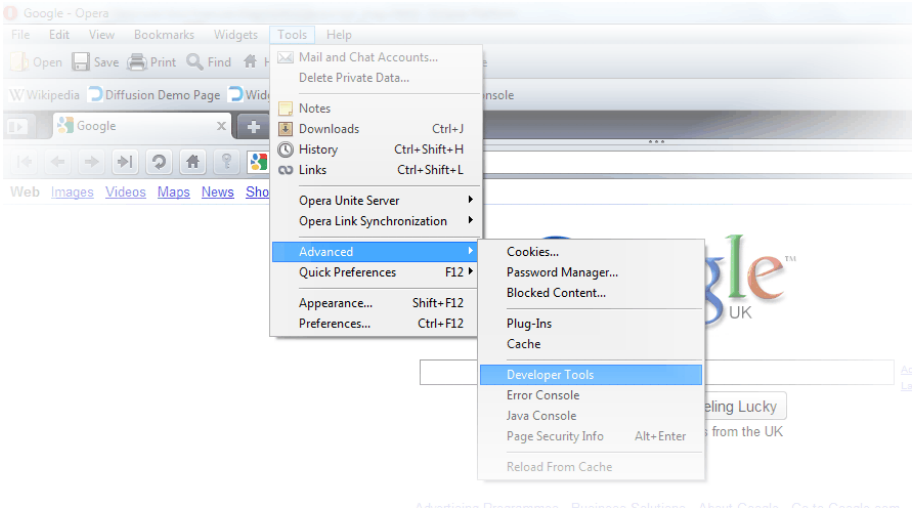
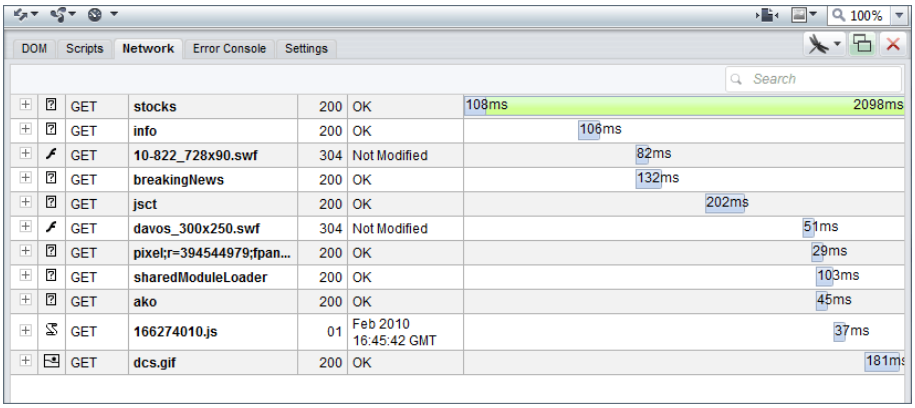


Figure 66: Opera console

Doing this opens the console that enables you to begin debugging.


DOM
Scripts
Network
Error Console
Settings
Search
GET stocks 200 OK 108ms 2098ms
GET info 200 OK 106ms
GET 10-822_728x90.swf 304 Not Modified 82ms
GET breakingNews 200 OK 132ms
GET jsct 200 OK 202ms
GET davos_300x250.swf 304 Not Modified 51ms
GET pixel;r=394544979;fpan... 200 OK 29ms
GET sharedModuleLoader 200 OK 103ms
GET ako 200 OK 45ms
GET 166274010.js 01 Feb 2010 16:45:42 GMT 37ms
GET dcs.gif 200 OK 181ms

Safari

Safari's console is Web Inspector, and functions in a very similar way to Chrome's Developer Tools. Profiling and Resources. Enabled it at first use

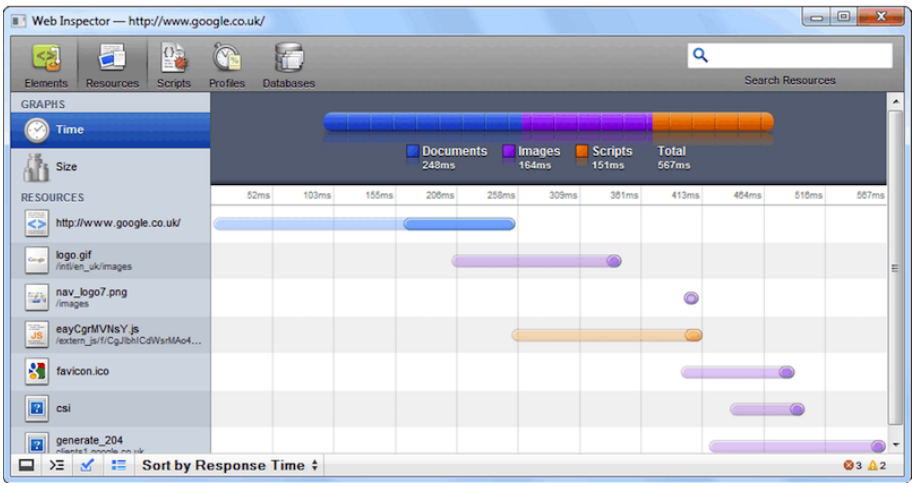


Figure 67: Safari's console

The DiffusionClient library has been compressed. If you want to use the DiffusionClient uncompressed version, use

```
<script type="text/javascript" href="lib/DIFFUSION/
diffusion.uncompressed.js"></script>
```

Flex and Flash diagnostics

For the Flash connectivity, it is important to got the policy files correct. If the policy file is not loaded or has the incorrect syntax or values, connectivity silently fails. You can get more information by following the steps below.

Diffusion also provides a debug-friendly version of the Flash client libraries: `diffusion-flex-debug-x.x.x.swc`, where *x.x.x* is the Diffusion version number, for example 5.1.0. You can use this to output additional debug information, such as line numbers, in any stack traces that you experience.

Flex/ActionScript diagnostics

1. Download the debug version for the Flash Player. Flash player debug versions are available at the following location: http://www.adobe.com/support/flashplayer/downloads.html
2. Create a `mm.cfg` file

The table below shows where to create a `mm.cfg` file

The following text is an example of entries in a `mm.cfg` file.

```
# Enables policy file logging
PolicyFileLog=1
# Optional; do not clear log at startup
PolicyFileLogAppend=0
ErrorReportingEnable=1
TraceOutputFileEnable=1
```

Viewing the log files

In the following directory there can be the following files: `flashlog.txt` and `policyfiles.txt`

Table 93: Location of the `flashlog.txt` file

Windows 95/98/ME	`%HOMEDRIVE%\%HOMEPATH%`
Windows 2000/XP	`C:\Documents and Settings\username`
Windows Vista	`C:\Users\username`
OS X	`/Library/Application Support/Macromedia`
Linux	`/home/username`

Table 94: Location of the `policyfiles.txt` file

Windows 95/98/ME/2000/XP	`C:\Documents and Settings\username\Application Data\Macromedia\Flash Player\Logs`
Windows Vista	`C:\Users\username\AppData\Roaming\Macromedia\Flash Player\Logs`
OS X	`/Users/username/Library/Preferences/Macromedia/Flash Player/Logs`
Linux	`home/username/Macromedia/Flash_Player/Logs`

Windows diagnostics

The .NET APIs and associated client applications produce logging information to log what they are doing and any problems that occur. Logging by default is to a file. The level of logging produced can be adjusted as required with high level logging under normal circumstances or very detailed logging to help with diagnosis of problems.

Configuring API logging

To configure the logging level for the .NET APIs, a developer can tailor the `DiffusionLogging` class for their own needs. The logger has a method for each level of logging (severe, warning, etc.).

Log levels

Supported log levels (in severity order) are as follows:

- Fatal
- Critical
- Error

- Warning
- Status
- Info
- Debug

Specifying any lower level includes logging for all levels above.

The method you use determines at what level of logging that message appears. So, by setting the SeverityThreshold to INFO:

```
DiffusionLogging.Logger.SeverityThreshold=Logseverity.Info;
```

This means that messages for only Info and Debug are logged.

Silverlight logging

There might be instances where a debug output viewing tool is unavailable; in these circumstances, you can 'redirect' all debug output to the Diffusion Debug Console for analysis. This is achieved by setting:

```
Debugging.RedirectDebugOutputToDiffusionConsole = true;
```

in the debug version of the Silverlight transport.

Debugging a publisher

Creating a project in Eclipse for debugging publishers.

After installing the Diffusion server, create a Java source directory (`src`) inside the Diffusion installation directory. Inside the `src` directory, create a `java` directory.

Create a new Eclipse Java Project and make the project root the src directory.

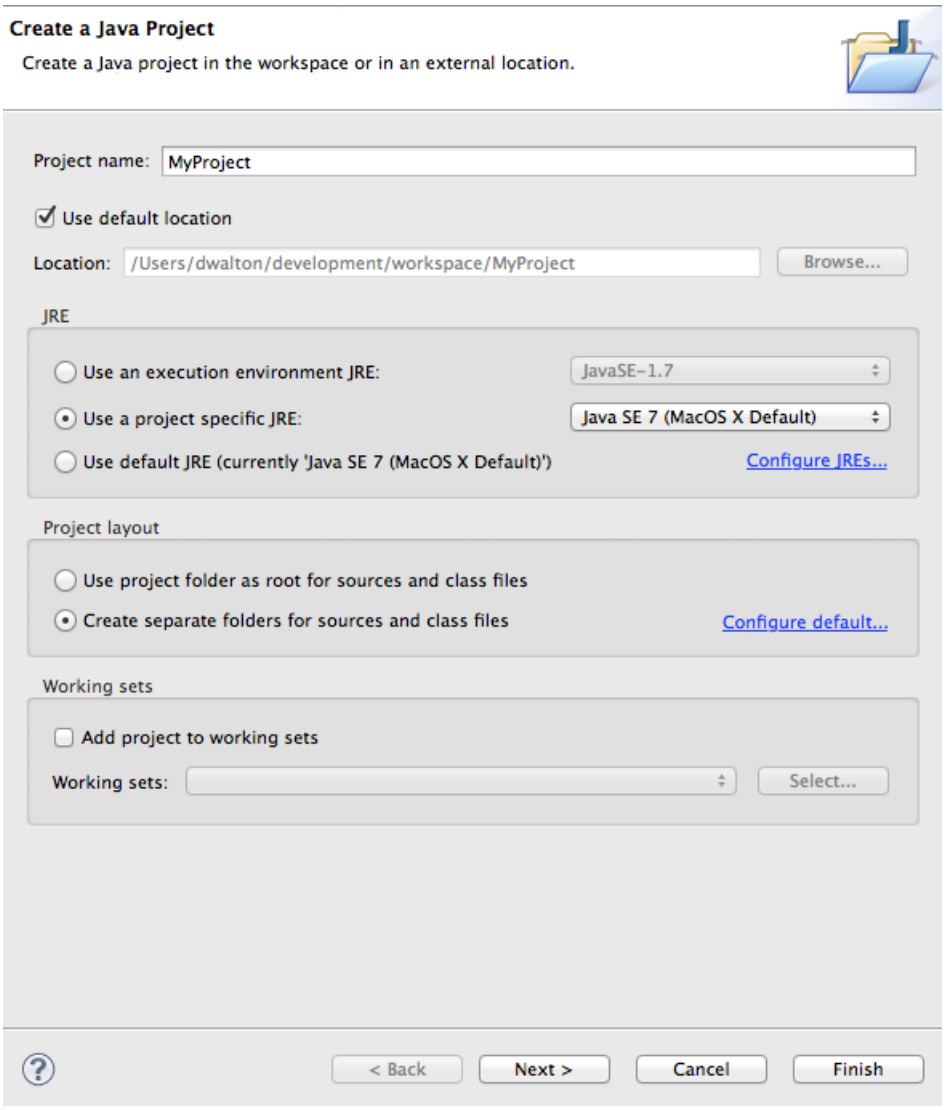


Figure 68: New Java Project

Press **Finish** to create the project. In the **Build Path** add the external jar `lib/diffusion.jar`.

In the `java` folder create the correct package hierarchy to suit your needs, and create the your publisher class that extends the class `com.pushtechnology.diffusion.api.publisher.Publisher`

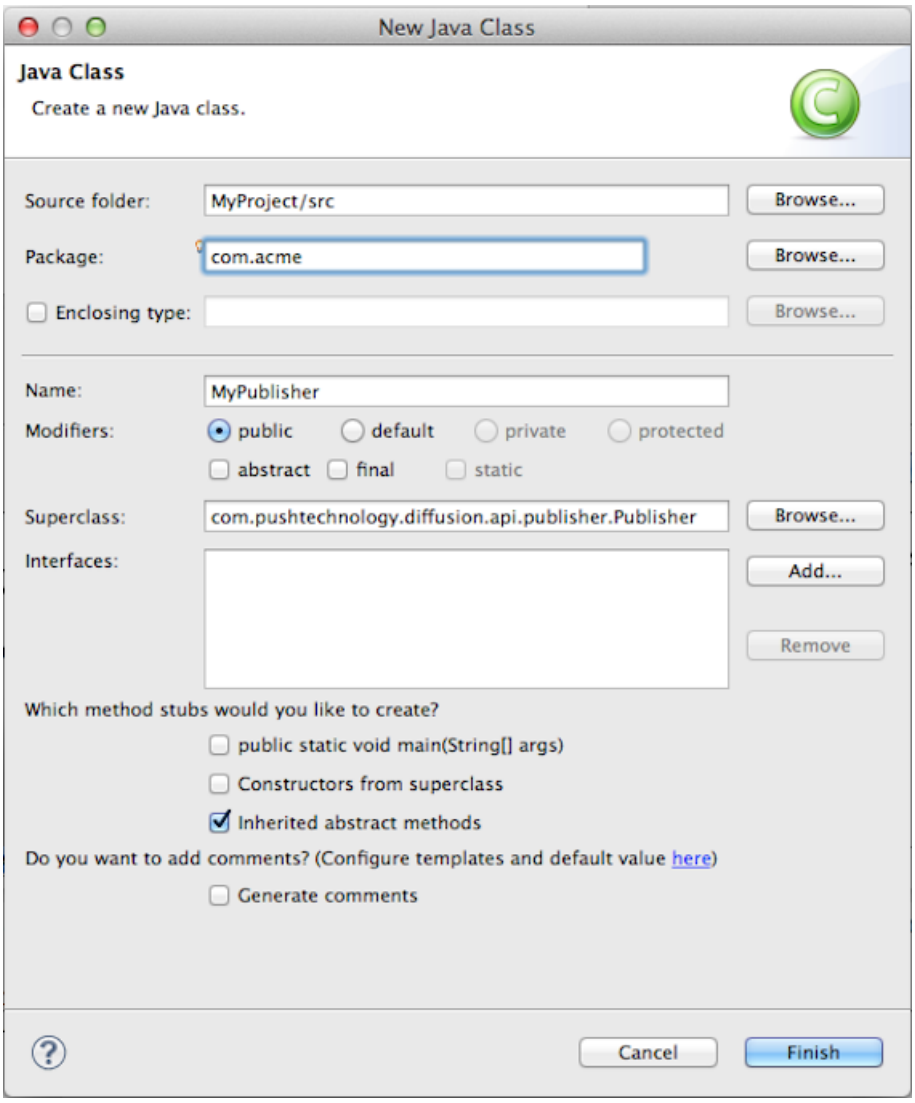


Figure 69: Creating a new Java class

Modify the properties

Modify `etc/Publisher.xml` to load the new publisher. Below is an example of the publisher definition for this publisher.

```
<publisher name="MyPublisher">
 <class>com.pushtechnology.publisher.MyPublisher</class>
</publisher>
```

Create a run target

Open the **Run Configurations** dialog in Eclipse and create a new run configuration, choose `com.pushtechnology.diffusion.Diffusion` as the main task. In the arguments tab, set the working directory to the `bin` directory of the Diffusion installation. Using the classpath tab, add to the user entries all jar files located in your installation's `ext` directory. Finally add your installation's `etc/` folder using the **Advanced** button. You can also add any external jar files that are required by your publisher.

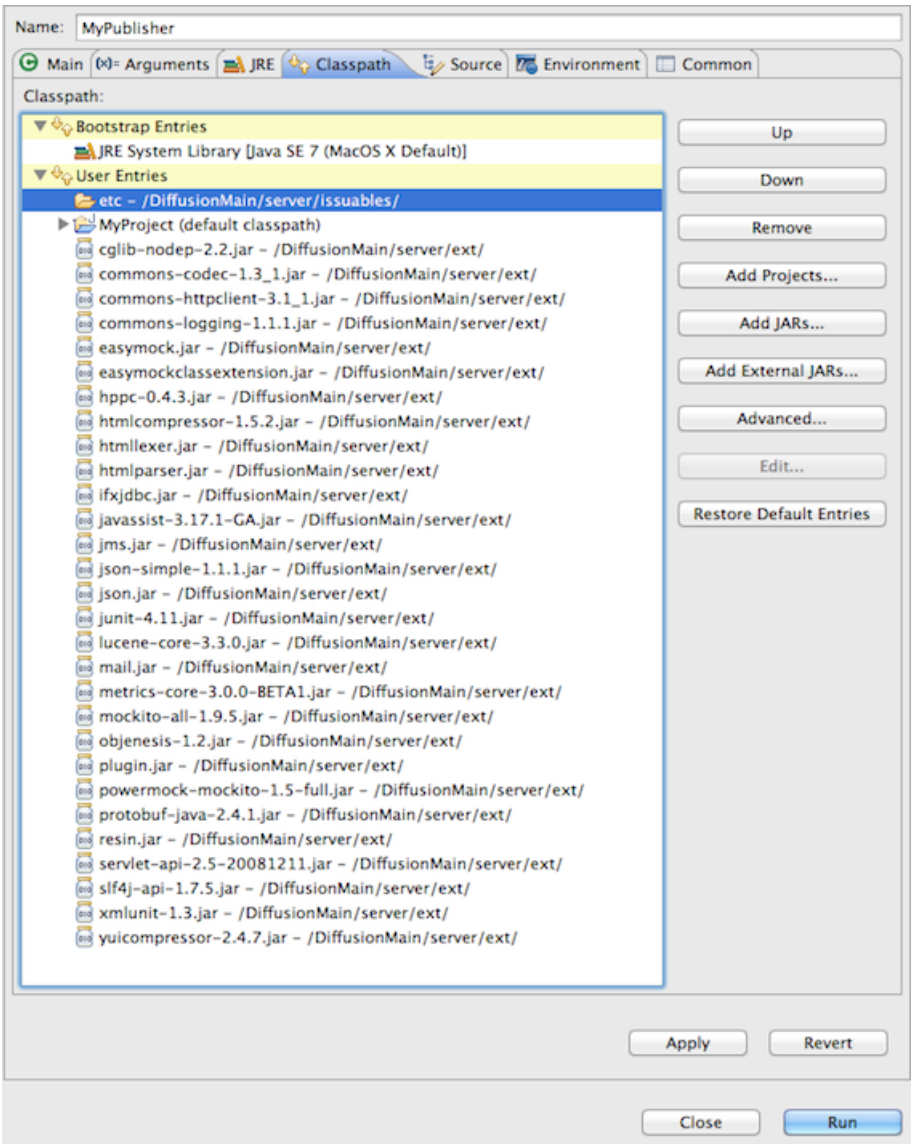


Figure 70: Example classpath entries

You can now set break points within the publisher. By default Eclipse creates a `bin` directory in the src directory, which contains the class files for your publisher. By setting the `etc/Server.xml` `usr.lib` entry to this `bin` directory, when you start Diffusion from the command line, your publisher is on the class path.

Log Messages

Log Reference

Log Messages

All Server Log Messages

Id	Message	Description
PUSH-000001	Attempt #{} to reconnect to JMS Server.	The JMS adapter is currently disconnected from a JMS server and has just failed another attempt to reconnect.

Id	Message	Description
PUSH-000002	Cannot stop the JMSAdapter: {}.	Something went wrong during initialisation of the JMS adapter, and there was an error when trying to clean up and stop the adapter.
PUSH-000003	Couldn't find JMSReplyTo destination: '{}'.	An attempt has been made to send a message with a JMSReplyTo destination, but the JMS adapter couldn't create the ReplyTo destination on the JMS server.
PUSH-000004	Discarding non-TextMessage from JMS: '{}'.	The JMS adapter only supports TextMessage types but it received another message type.
PUSH-000005	Exception from JMS provider '{}'.	The JMS adapter received notification of an exception from the JMS server. Typically, this happens when the connection between the adapter and the server has been terminated.
PUSH-000006	Extra JMS header found, ignoring: '{}'.	JMS messages can have user-defined headers as key/value pairs. Diffusion messages are effectively "value" only, so to map user headers from Diffusion to JMS there must be an even number of them; these are artificially grouped as key/value pairs in the JMS message.
PUSH-000007	Failed to find topic binding for '{}'.	An attempt has been made to send a reply from the JMS adapter back to the JMS server using the DiffusionReplyTo header, but the adapter was unable to determine the corresponding JMSReplyTo destination.
PUSH-000008	Failed to generate delta message '{}'.	A message has been received from the JMS server and a delta message needs to be sent, but the update or publish of the topic data associated with the Diffusion topic failed.
PUSH-000009	Failed to generate ITL message '{}'.	An error occurred while generating an ITL message for a client after a message has been received from a JMS queue.
PUSH-000010	Failed to get message content of message to JMS, topic is '{}'.	Failed to read/parse the content of a Diffusion message while translating it to a JMS message for sending.
PUSH-000011	Failed to send message to JMS destination '{}', exception: {}.	An error occurred while sending a message from the JMS adapter to the JMS server.
PUSH-000012	Failed to start JMS Adapter {}.	The JMS adapter was unable to start.

Id	Message	Description
PUSH-000013	Maximum number '{}' of reconnection attempts reached, JMSAdapter disconnected.	The adapter has exhausted the maximum number of configured connection attempts with the JMS server and has given up.
PUSH-000014	Received TextMessage for unknown destination: '{}'.	A message was received by the JMS adapter from the JMS server, but was unable to determine which Diffusion topic it should be mapped to.
PUSH-000015	JMS Server disconnect detected, attempting reconnection.	The JMS server has been disconnected from the JMS adapter and the adapter is attempting to reconnect.
PUSH-000016	JMS Server reconnected after '{}' attempts.	The JMS adapter has managed to (re)connect to the JMS server.
PUSH-000017	Unable to create a subscription to '{}', exception is {}.	The JMS adapter failed to create a subscription to the JMS destination associated with the given topic name.
PUSH-000018	Unable to create new topic '{}'.	The JMS adapter has received a message from the JMS server and needs to create a corresponding Diffusion topic, but was unable to.
PUSH-000019	Unable to extract headers from JMS message: '{}'.	Something went wrong while attempting to convert JMS message headers to Diffusion message headers.
PUSH-000020	Unable to get payload from JMS message '{}'.	The JMS adapter has received a message from the JMS server, but is unable to read the content.
PUSH-000021	Unable to restart JMS Server connection.	After an exception from the JMS server, the JMS adapter attempted to reconnect but was unable to.
PUSH-000022	Exception has been thrown by the ServerAckListener.messageNotAcknowledged method of {}.	The specified listener methods has thrown an exception which has been logged but otherwise ignored.
PUSH-000023	Client Auto Failover failed.	A connection to a server has failed, and the auto failover process has encountered an error.
PUSH-000024	Mime extension '{}' was already mapped to '{}' : overwriting map with '{}'.	A configured Mime extension was already mapped to another value but has now been remapped to a new value. This indicates duplicate mime extension values in the mimes configuration.
PUSH-000025	Connection attempt failed with {} - trying next server.	An attempt to connect to a server has failed but more than one possible server has been specified and so the next is being tried.

Id	Message	Description
PUSH-000026	Requested receive buffer size for HTTP Proxy connection not allocated, requested: {}, allocated {}.	Diffusion tried to set the size of the TCP receive buffer for a socket connection to an HTTP proxy, but the actual number of bytes allocated was different.
PUSH-000027	Requested send buffer size for HTTP Proxy connection not allocated, requested: {}, allocated {}.	Diffusion tried to set the size of the TCP send buffer for a socket connection to an HTTP proxy, but the actual number of bytes allocated was different.
PUSH-000028	An exception has been thrown by the {} method in {}.	An exception has been thrown by an implementation of the specified listener interface. The exception is logged but has no further impact.
PUSH-000035	Unable to send fetch reply to Client '{}'.	An error occurred while sending a fetch reply message back to a client.
PUSH-000036	Failure fetching state from '{}' of '{}'.	While handling a fetch request for a topic with topic data, an error occurred while generating a topic load message.
PUSH-000037	Subscription Handler '{}' specified for Routing Topic '{}' - ignored.	A subscription handler has been specified for a Routing Topic which is not allowed therefore the handler has been ignored.
PUSH-000038	Exception caught from TopicDeletionListener '{}' topicDeleted method.	An exception has been thrown from a call to a TopicDeletionListener.topicDeleted method. The exception has been logged but has no other effect.
PUSH-000039	Exception caught from TopicTreeListener '{}' {} method.	An exception has been thrown from a call to a TopicTreeListener method. The exception has been logged but has no other effect.
PUSH-000040	WhoIs connection failure limit exceeded - not resolving.	Have attempted to connect to the WhoIs provider 5 times but failed therefore there will be no WhoIs service.
PUSH-000041	Failure to connect to WhoIs provider at {}:{} - will retry 5 times.	Unable to connect to the WhoIs provider indicated - will try 5 more times before giving up.
PUSH-000042	Failed to enumerate files in usr.lib directory '{}'.	An error has occurred while traversing the files within the given directory.
PUSH-000043	Attempted to load duplicate jar file '{}'.	An attempt was made to load the given file more than once.
PUSH-000044	Entry '{}' in server configuration is not a directory.	The given file-system name should identify a directory.

Id	Message	Description
PUSH-000045	Attempt to queue message '{}' for client '{}' when ACK timeout has already expired.	An acknowledgement would be queued, in response to a message requiring an acknowledgement, however the ACK timeout has already expired and is therefore probably too short.
PUSH-000046	Client '{}' closing - {}.	The given client is closing.
PUSH-000047	Client '{}' closing - {}, exception='{}'.	The given client is closing due to an exception.
PUSH-000049	Error subscribing Client '{}' to Topic '{}': {}.	An error has occurred while trying to subscribe the specified Client to the specified Topic.
PUSH-000050	Client '{}' sent command message to Topic '{}' which does not have command-based Topic Data. Message is '{}'.	Command messages can only be processed by topics with related command-based Topic Data.
PUSH-000051	Failed to apply Client change event : '{}'.	An error occurred while applying a change to the given client.
PUSH-000052	Exception in ACK listener '{}' messageNotAcknowledged method.	An error occurred while informing a listener that a dispatched message has not been acknowledged.
PUSH-000053	Failed to queue background close of client '{}'.	An attempt to queue the close of a client has failed and therefore the client has been closed in the current thread.
PUSH-000054	Client '{}' tried to fetch invalid topic-selector '{}'.	The topic-selector expression supplied by a Client for a fetch request was invalid.
PUSH-000055	Client Group "'{}'" failed to send message to '{}'.	In sending a message to a Client Group, transmission to the given individual client encountered an error.
PUSH-000056	User licence limit breached, connection rejected.	The maximum number of licensed connections has been exceeded.
PUSH-000057	Server is not licensed to accept distributed connections.	This Diffusion server is not licensed to accept connection from other Diffusion servers.
PUSH-000058	Server is not licensed to accept mobile connections.	The licence does not allow for Diffusion Mobile-to-Server connections.
PUSH-000059	Client '{}' sent message to Topic '{}' to which it is not subscribed.	Diffusion mandates that a client must be subscribed to a topic before it can send messages to it.
PUSH-000060	Unable to subscribe Client '{}' to Topic '{}' - Timed out attempting to acquire lock.	A timeout has occurred while attempting to acquire a lock on a Topic in order for a Client to subscribe to it.

Id	Message	Description
PUSH-000061	Failed to subscribe Client '{}' to Topic(s) '{}' - Invalid Topic name or Selector.	The Client sent an invalid Topic selector pattern for subscription.
PUSH-000062	Failed to schedule future close of Client '{}'.	It was not possible to schedule the future close of a Client that is being kept alive for reconnect purposes.
PUSH-000063	Unable to send fetch reply message to Client '{}' for Topic '{}'.	A failure occurred while trying to send a fetch reply to a Client.
PUSH-000064	Failed to unsubscribe Client '{}' from '{}' - Invalid Topic name or selector.	The Client sent an invalid Topic selector pattern for unsubscription.
PUSH-000065	Failure processing message from Server.	An error has occurred at the Client end of a connection to a Server while processing a message from the Server.
PUSH-000066	{} closing due to send failure {} {}.	An error occurred while and Event Publisher connection was sending messages.
PUSH-000067	Exception in EventPublisherListener.{} method.	Exception thrown by EventPublisherListener method.
PUSH-000068	Timed out waiting for all messages to be sent.	The EventPublisher connection has been closed while there were messages pending to be sent. The EventPublisher was unable to send those messages in a timely fashion, and they may now be lost.
PUSH-000069	Unexpected error while sending message.	An error occurred when an EventPublisher tried to send a message.
PUSH-000070	Invalid connection from '{}' ('{}').	An invalid connection has been attempted from the given address and location.
PUSH-000071	Unable to rebuild fragmented message.	An error occurred while trying to reassemble message fragments into the original message.
PUSH-000072	Connector '{}' created for connections of type {}, listening on {}.	The given connector was created to support the given types of connection.
PUSH-000073	Connector '{}' does not support Clients or Event Publishers.	A connector has received an attempted connection which indicates that it is either a Diffusion Client or Event Publisher but the connector supports connections from neither.
PUSH-000074	Connector '{}' received an HTTP connection but does not support HTTP.	Connector has received an HTTP connection attempt but does not support HTTP connections (i.e. does not define a valid web server).
PUSH-000075	Unable to start Connector '{}'.	An error occurred while trying to start a connector.

Id	Message	Description
PUSH-000076	Unable to start Connector '{}', permission denied.	The OS denied permission for a resource used by this connector. Often this relates to reserved ports (less than 1024) on Unix based operating systems.
PUSH-000077	HTTP Method '{}' is not supported in request from '{}'. Request: '{}'.	An HTTP client has requested a method that is not supported by Diffusion.
PUSH-000078	HTTP Processing error, request from '{}'.	An error occurred in the processing of an HTTP request from the given client.
PUSH-000079	Connector '{}' received a Policy File request which is not supported.	Connector has received a policy file request but does not support such requests (i.e. does not have a policy file configured).
PUSH-000080	Non-SSL connection attempted to connector '{}' which only supports SSL-type connections.	An attempt has been made to make a non secure (SSL) connection to a connector that is configured to only accept secure connections.
PUSH-000081	Connector '{}': no connection - SSL handshake error: {}.	A secure (SSL) connected was attempted but the SSL handshake failed.
PUSH-000082	SSL connection attempted but connector '{}' does not support SSL.	An attempt has been made to make a secure (SSL) connection via a connector that does not support SSL (i.e. does not define a key store).
PUSH-000083	Connector '{}' received a Client connection but does not support Clients.	The given connector is not configured to receive and handle Client connections. The connector may be dedicated to Event Publisher connections, for example.
PUSH-000084	Connector '{}' received an Event Publisher connection but does not support Event Publishers.	The given connector is not configured to receive and handle Event Publisher connections. The connector may be dedicated to Client connections, for example.
PUSH-000085	{}: Requested input buffer size could not be allocated, requested: '{}' allocated: '{}'.	The receive buffer of the socket was assigned a different amount of memory than requested. This is configured by the input buffer size. The configured input buffer size is a hint to the operating system, you should refer to you OS documentation for any socket buffer limits. This can have performance implications.
PUSH-000086	{}: Requested output buffer size could not be allocated, requested: '{}' allocated: '{}'.	The send buffer of the socket was assigned a different amount of memory than requested. This is configured by the output buffer size. The configured output buffer

Id	Message	Description
		size is a hint to the operating system, you should refer to you OS documentation for any socket buffer limits. This can have performance implications.
PUSH-000087	Connector '{}' received an unidentified connection request [{}] from {}.	An unidentified connection attempt has been made via a connector. The hexadecimal representation of the initial bytes of the connection request are logged in order to aid in identifying the origin.
PUSH-000089	Failed to update Child List Topic Data for Topic '{}'.	An unforeseen error occurred while updating this ChildListTopicData.
PUSH-000090	An exception has been thrown from CustomTopicDataHandler.{} method.	The custom TopicData implementation threw an exception in the given method.
PUSH-000091	Client '{}' sent invalid topic selector pattern '{}' to Topic Notify Topic.	A Client has sent an invalid Topic name or selector pattern to a Topic Notify Topic selection.
PUSH-000092	An exception has been thrown from TopicNotifyTopicListener.{} method in '{}'.	The Topic Notify Topic Listener threw an exception in the given method.
PUSH-000093	Topic Notify Select Request Mode '{}' from Client '{}' is invalid.	Client has sent an invalid select request mode to a Topic Notify Topic.
PUSH-000094	{} was unable to notify Client '{}' of Topic '{}' notification event - Client closing.	A failure has occurred notifying a Client of a Topic addition, removal or update - the client will be closed.
PUSH-000095	Unrecognised notification type '{}'.	The notification request holds unexpected content.
PUSH-000096	Client '{}' request to lock Paged Data '{}' timed out.	Unable to action a request from a given client on the specified data as it was not possible to acquire a lock within the timeout period.
PUSH-000097	Paged Topic '{}' Unable to process command '{}' from client '{}'.	A Paged Topic was unable to process a command from a Client.
PUSH-000098	Failure sending a Paged Topic status message to Client '{}'.	A failure occurred while sending a Paged Topic status message to the specified Client.
PUSH-000099	Failure sending a Paged Topic update message to Client '{}'.	A failure occurred while sending a Paged Topic update message to the specified Client.
PUSH-000100	Command Topic Data '{}': received invalid command message '{}'.	A Client has sent an invalid command message to Command Topic.
PUSH-000142	Error removing routing data {} from {}.	A failure occurred while trying to remove a topic from a routing topic.
PUSH-000143	Routing Topic {} failed to route message to client '{}': {}.	A Topic with Routing Topic Data has failed to route a published message to a specific Client.

Id	Message	Description
PUSH-000144	Routing Topic failed to subscribe Client '{}' to Topic '{}'.	A Routing Topic failed to subscribe a Client to a Topic.
PUSH-000145	An exception has been thrown from ServiceTopicListener.{} method in '{}'.	The Service Topic Listener threw an exception in the given method.
PUSH-000146	Service Request error for Client '{}' on Service '{}', request id '{}', error '{}'.	A request from a Client to a Service Topic has failed and it was not possible to report the failure back to the Client, possibly because it is closed.
PUSH-000147	Failure in TopicDataListener '{}' topicDataChanged method.	An exception has been thrown by an implementation of the TopicDataListener.topicDataChanged method. The exception is simply logged and has no other effect.
PUSH-000148	Unable to lock Topic {} in order to notify Routing Topic Data instances of Topic removal.	A timeout occurred while waiting to lock the specified Topic in order to notify instances of RoutingTopicData of the Topic removal. It is possible that this error could leave one or more instances of RoutingTopicData in an inconsistent state.
PUSH-000149	{} unable to publish to {}.	An update to a Topic has resulted in the publishing to other Topics (e.g. when updating a Topic that has SlaveTopicData instances pointing at it) and a failure has occurred publishing a message to one of those other Topics.
PUSH-000150	Message '{}' received from '{}' by '{}' has not been handled.	A command Topic that does not handle Messages from Clients has received a message from a Client and ignored it.
PUSH-000151	Unable to get JMX MBean attribute.	It was not possible to retrieve an attribute from a JMX MBean.
PUSH-000152	Unable to load shutdown hook class '{}': {}.	An error occurred loading the given 3rd party class.
PUSH-000153	Unable to load startup hook class '{}': {}.	An error occurred loading the given 3rd party class.
PUSH-000154	Caught JVM shutdown.	The JVM shut down unexpectedly.
PUSH-000155	No connectors have been configured. Creating default connector listening on port {}.	No connectors have been configured therefore a default connector listening on port 8080 has been created with all default values.
PUSH-000156	No Multiplexer Configuration. Creating default configuration.	No Multiplexer configuration has been found therefore a default Multiplexer configuration will be assumed.

Id	Message	Description
PUSH-000157	No queue definitions configured. Creating DefaultQueue.	No queues have been configured therefore a default queue definition has been assumed.
PUSH-000158	No thread pools configured. Created new pool definition called '{}'.	No thread pools were configured therefore a default one has been added.
PUSH-000159	The maximum message size is {} bytes.	The maximum message size has been established.
PUSH-000160	No inbound pool has been configured - using '{}'.	No inbound thread pool has been configured therefore the first configured thread pool has been assumed to define the inbound pool.
PUSH-000161	Diffusion - Removing Publishers.	The server is exiting, and all publishers within are being unloaded.
PUSH-000162	Running shutdown hook '{}'.	The given 3rd party class is being executed at server shutdown.
PUSH-000163	Running startup hook '{}'.	The given 3rd party class is being executed at server startup.
PUSH-000164	Diffusion Server not started.	Diffusion Server failed to start.
PUSH-000165	Diffusion Server started.	Diffusion started successfully.
PUSH-000166	Diffusion Server '{}' starting.	The Diffusion Server is starting.
PUSH-000167	Diffusion Stopped.	The Diffusion server has been stopped.
PUSH-000168	Diffusion Stopping, reason='{}' by administrator='{}'.	Diffusion is processing a shutdown request.
PUSH-000169	Diffusion - Stopping Connectors.	Connectors are being stopped during a shutdown of Diffusion.
PUSH-000170	Event Publisher {} sent Command message for Topic '{}' which does not have command based Topic Data.	An Event Publisher has sent a command message to a Topic that is not a Command Topic. The message has been ignored.
PUSH-000171	Event Publisher {} sent message for Topic '{}:' which does not exist.	An Event Publisher sent a message for a Topic that does not exist. The message has been ignored.
PUSH-000172	Unable to start StatisticsHandler for Event Publisher '{}'.	An error occurred while registering an Event Publisher with the statistics framework.
PUSH-000173	Exception in '{}' of Listener '{}'.	An exception occurred while processing an internal asynchronous event.
PUSH-000174	Unable to submit event '{}' to thread pool '{}' for execution.	A failure has occurred while submitting a notification event to the specified thread pool for execution.
PUSH-000175	SHA1 algorithm not found, unable to respond to WebSocket request.	The WebSocket protocol requires the SHA1 algorithm, but it was not found. This can only happen with

Id	Message	Description
		an incomplete or incompatible Java installation.
PUSH-000177	Received invalid message '{}' from '{}': {}.	The server module dealing with Introspector clients has received an invalid message.
PUSH-000178	Exception publishing message on topic '{}'.	The server module dealing with Introspector clients caught an exception sending a message to the given client.
PUSH-000180	Topic '{}' may not be removed by Introspector for client {}.	The given client has attempted to remove the given irremovable topic.
PUSH-000181	Topic '{}' not removed for client '{}': {}.	The given topic was not removed for the given client and reason.
PUSH-000182	Topic '{}' removed by Introspector client '{}'.	The given topic was removed for the given client.
PUSH-000183	Message channel error: '{}' caused by '{}'.	A communication error has occurred on a Message Channel.
PUSH-000184	Connector '{}:' has no more Acceptors - Stopping.	An acceptor for a connector has failed, and there are no more acceptors available to service connection requests therefore the connector is stopping.
PUSH-000185	Failed to accept connection for acceptor '{}' on connector '{}'.	This can occur when a socket connection has been made to Diffusion, but a failure occurred while initialising it.
PUSH-000187	Second attempt to write output, try increasing timeout or output buffers '{}'.	Diffusion has tried to obtain a selector for writing and failed and is trying again. If this problem persists, try increasing the write timeout or the output buffer size.
PUSH-000188	Selector Thread Failure {} - Closing.	An error has occurred in a selector thread and the thread is closing.
PUSH-000190	SSL unwrap error: '{}'.	An error occurred while decoding SSL-wrapped data.
PUSH-000191	Connector '{}:' - Unable to accept connection.	An exception occurred while attempting to accept a socket connection.
PUSH-000193	SSL Keystore loaded from '{}'.	SSL Keystore has been loaded from the specified location.
PUSH-000194	JMX cannot expose '{}' as a topic as dimensions exceed 2.	Only 1 and 2 dimensional arrays can be mirrored as a Topic.
PUSH-000195	JMX: Cannot register object '{}' at MBean ObjectName '{}': {}.	An error occurred registering the given object with the given JMX ObjectName with the JMX server.
PUSH-000196	'{}:' failed to update Diffusion licence.	An error occurred while attempting to update the licence file.

Id	Message	Description
PUSH-000197	Unable to create directory '{}' for the public certificates.	The HTTP service which can be used for updating the licence needs to be able to store the licence in a directory, but that directory didn't exist and couldn't be created.
PUSH-000198	Licence is invalid.	The Licence is invalid.
PUSH-000199	Licence is not valid for this version of Diffusion (licence='{}' vs '{}').	The licence file does not match this version of Diffusion.
PUSH-000200	Product '{}' not licensed for this address scheme.".	A product is not licensed for either IP address or MAC addressing schemes.
PUSH-000201	Soft user limit for {} exceeded ({}/{}). Hard limit at {}.	Once the number of connections for a product reaches its soft limit, a warning is emitted. The product may cease to function if this number reaches the hard limit.
PUSH-000202	Licensor: Only '{}' days remaining.	The installed licence file is nearing expiration. A new licence will be required soon in order for Diffusion to continue running.
PUSH-000203	Licence has expired.	The Diffusion licence has expired.
PUSH-000204	Licence hard user limit for {} exceeded ({}/{}).	Once the number of client connections has reached or exceeded a the hard limit, the product may be disabled.
PUSH-000205	Licence for Introspector has expired.	Licence for Introspector has expired.
PUSH-000206	Licensor stopping Diffusion! Invalid licence.	The Diffusion licence is invalid.
PUSH-000207	Licence check failed : Unable to check Server Side Addressing for product [IP] '{}'.	During the licence check, Diffusion attempts to check the server's IP addresses but it was unable to do so.
PUSH-000208	Licence check failed : Unable to check Server Side Addressing for product [MAC] '{}'.	Diffusion is licensed to run on a server with a specific MAC address, but an error occurred while looking for MAC addresses on the server.
PUSH-000209	Licence check failed : Unable to match MAC address for ['{}'].	The product is licensed for a particular MAC address, but the server does not have this MAC.
PUSH-000210	Licence check failed : Unable to parse CIDR Address: '{}'.	The product is licensed to run on machines with IP addresses matching a CIDR expression, but the expression could not be parsed. Contact Push Technology for an updated licence file.
PUSH-000211	Licence version is empty, this is not allowed.	Licence version is empty, this is not allowed. Contact Push Technology for a replacement licence file.
PUSH-000212	Log level set to '{} for '{}'.	The logging level of the specified log file has been changed as indicated.

Id	Message	Description
PUSH-000213	Unable to register '{}' with Management Server.	An error occurred while registering the specified object with the JMX Management server.
PUSH-000214	Invalid JMX credentials.	The supplied JMX credentials are incorrect.
PUSH-000215	The management service is disabled.	The Diffusion Management Service is configured to be disabled.
PUSH-000216	The management service has started.	The Diffusion Management Service has started.
PUSH-000217	Unable to Start Management Service.	An error occurred while starting the Diffusion Management Service .
PUSH-000218	Unable to register {} with Management Server - No Management Support.	No management has been configured and therefore it is not possible to register objects. Only the first failure to register an object for this reason will be logged.
PUSH-000220	Unable to schedule fragmented message for sending.	A message is being sent in fragments with a delay between each fragment, but Diffusion was unable schedule delivery of the next fragment.
PUSH-000221	Fragment send error.	An error occurred while sending a fragment of a fragmented message.
PUSH-000222	Spent too long trying to send fragmented message with correlation-id '{}', cancelling.	Diffusion has taken too long trying to send fragments of a fragmented message, so the message will be cancelled.
PUSH-000223	Message queue limit {} reached while publishing fragmented message.	A client outbound queue had reached its limit as a result of a message fragmentation operation.
PUSH-000224	Failed to schedule fragmented message for sending.	Unable to schedule the fragmentation of a message before sending.
PUSH-000225	Error duplicating multiplexer event message.	An error has occurred duplicating a multiplexer event message.
PUSH-000226	Multiplexer error while processing client '{}'.	A multiplexer tried to send messages for a client, but failed.
PUSH-000227	Multiplexer event is delayed as the queue is full.	An event-driven multiplexer couldn't queue an event due to the queue being full but is continuing to try to do so.
PUSH-000228	Multiplexer event is significantly delayed as the queue is full.	due to the queue being full, an event-driven multiplexer couldn't queue an event after a significant amount of time, but it is continuing to try to do so.
PUSH-000229	Error while handling a multiplexer event.	Error while handling a multiplexer event.

Id	Message	Description
PUSH-000230	Failed to schedule event.	An event-driven multiplexer tried to schedule an event for some time in the future, but wasn't able to do so.
PUSH-000231	Possible latency warning from '{}' [{}ms].	The configured latency warning time has been exceeded while a multiplexer is sending to a client.
PUSH-000232	No multiplexers configured.	No multiplexer definitions have been configured so a default has been assumed.
PUSH-000233	Event-driven multiplexer '{}' started .	An event-driven multiplexer has been started.
PUSH-000234	Properties loaded: '{}'.	The specified properties file has been loaded.
PUSH-000235	Properties reloaded: '{}'.	The specified properties file has been reloaded.
PUSH-000236	Cannot establish JMX attribute '{}' for MBean '{}': {}.	The given MBean/Attribute pair is unsupported by Diffusion.
PUSH-000237	Cannot get JMX attribute '{}': {}.	An error occurred while fetching the value of the given MBean attribute.
PUSH-000238	JMX Attribute '{}' for MBean '{}' is unsupported: {}.	The given MBean/Attribute pair is unsupported by Diffusion.
PUSH-000239	Cannot create topic for MBean {} {}.	It was not possible to create a topic to mirror the content of the given JMX MBean.
PUSH-000240	JMX Adapter unable to get topic '{}': {}.	An error in the JMX Adapter while trying to get the given topic from the topic-tree.
PUSH-000241	Cannot remove topic for MBean {}.	It was not possible to remove a topic that mirrors the content of the given JMX MBean .
PUSH-000242	JMX Configuring property '{}'={}.	The JMX Adapter configuration is being applied.
PUSH-000243	JMX Cannot parse property '{}': {}.	An error occurred reading the given property.
PUSH-000244	JMX Exception interacting with MBean '{}': {}.	An error occurred creating the tree of topics to represent a JMX MBean.
PUSH-000245	JMX Exception while updating topic '{}': {}.	An error occurred while polling an MBean attribute that the given topic reflects.
PUSH-000246	Caught exception removing notification listener from JMX '{}'.	Diffusion is no longer listening to notification from the given MBean, and this has caused a problem.
PUSH-000247	Cannot find MBean '{}' to listen for notifications.	Diffusion is attempting to add a notification to the given MBean, and this has caused a problem.

Id	Message	Description
PUSH-000248	Listener for Mbean '{}' not found.	The notification listener being removed from the given MBean is not currently registered with it.
PUSH-000249	Cannot send message given notification object {}.	A JMX notification was detected and serialised as a message, but it was not possible to send the message.
PUSH-000250	Unexpected notification object '{}'.	A JMX notification of an unexpected type was detected.
PUSH-000251	Unexpected notification type '{}'.	MBeanServerNotification received that was neither about an MBean registering or deregistering.
PUSH-000252	Unable to construct topics for DiffusionPublisher Log Handler: {}.	The Diffusion publisher log handler encountered an error trying to create its topics.
PUSH-000253	Exception caught in {} method of Authorisation Handler {}.	An exception has been thrown by the specified method of the user written Authorisation Handler - the exception has been logged and the requested action rejected.
PUSH-000254	Authorisation manager started with authorisation handler class '{}'.	Authorisation Manager has been started with the specified handler class.
PUSH-000255	Publisher {} connected to Server {} as {}.	A Publisher has successfully connected to another Diffusion server as a client.
PUSH-000256	Deploying publishers compiled for Diffusion version '{}'.	Publishers compiled for Diffusion Version specified are being deployed.
PUSH-000257	Publisher '{}' did not handle message from Server '{}'.	A Publisher has received a message from a server but has not provided an implementation for the messageFromServer method.
PUSH-000258	Publisher '{}' did not handle server rejected credentials for '{}'.	The default (unimplemented) serverRejectedCredentials method of a Publisher has been called because credentials it has sent to a Server have been rejected.
PUSH-000259	Publisher '{}' did not process message from Event Publisher '{}'.	A Publisher has received a message from an Event Publisher but has not implemented the messageFromEventPublisher method in order to handle it.
PUSH-000260	Diffusion-Version not present in manifest file, not deploying.	Diffusion-Version not present in DAR manifest file so the Publisher(s) will not be deployed.
PUSH-000261	Error deploying Publisher from file '{}'.	An exception has occurred while trying to deploy a Publisher from a DAR file.
PUSH-000262	Error writing file for deployment: {}.	When hot-deploying publishers, the contents of DAR files must be

Id	Message	Description
		extracted to disk. Check for write permissions to the deployment directory and that there is sufficient free disk space.
PUSH-000263	Exception caught in Publisher '{}' in method '{}'.	A Publisher implementation has thrown an exception in the given method. The exception has been logged but otherwise ignored.
PUSH-000264	Exception caught in publisher '{}' in systemStarted method.	Publishers declared in etc/Publishers.xml can be informed that Diffusion is ready by overriding the systemStarted() method. This message is emitted if an Exception is thrown by the Publisher's systemStarted() method.
PUSH-000265	Failed to load publisher '{}'.	Diffusion was unable to load a publisher, most likely due to a problem with the publisher's configuration files.
PUSH-000266	This dar file is for Diffusion version {} or greater. Not deploying on Diffusion version {}.	The META-INF/MANIFEST.MF of the DAR file contains the "Diffusion-Version" attribute that specifies with which versions of Diffusion it is compatible. The DAR is marked as incompatible with this version of Diffusion and will not be deployed.
PUSH-000267	Introspector Publisher defined, but not licensed.	Diffusion is attempting to start the publisher which is required for the Introspector, but the licence file does not allow for the Introspector to be run.
PUSH-000268	Publisher '{}' is not stoppable, not undeploying.	An attempt has been to undeploy a publisher, but it's isStoppable() method returns false. The publisher will not be undeployed.
PUSH-000269	JMX adapter threw exception while starting: {}.	An unexpected exception prevented the JMX adapter from starting.
PUSH-000270	Configured not to poll MBeans.	The JMX adapter is configured not to poll those JMX MBean attributes with a matching topic subscription.
PUSH-000271	JMX adapter configured not to start.	The JMX adapter is configured not to start, and therefore will be unavailable.
PUSH-000272	Polling MBeans every {} milliseconds.	The JMX adapter is configured to configured to poll required MBean attributes for the given frequency.
PUSH-000273	JMX adapter starting.	The JMX adapter has begun construction.
PUSH-000274	DeploymentMonitor: The dar file '{}' has not been fully undeployed.	An error occurred during an attempt to undeploy a publisher, which had

Id	Message	Description
	The publisher(s) {} have not been undeployed. The dar file will be ignored from now on.	been deployed using the "hot deploy" feature. This could happen if a DAR contains multiple publishers, but not all of them return "true" from the isStoppable() method.
PUSH-000275	Publisher '{}' received notification of non-acknowledgement of message '{}' by {} clients.	A publisher has received a notification that one or more Clients have not acknowledged a message requiring acknowledgement, but has not implemented the messageNotAcknowledged method to handle it.
PUSH-000282	Removed Publisher '{}'.	The given Publisher has been removed from the server.
PUSH-000283	Publisher to Server connection '{}' closing due to send failure: {} {}.	A connection between a Publisher and a Diffusion Server is being closed due to a failure while sending messages to the Server.
PUSH-000284	Publisher '{}' unable to connect to Server '{}' - no recovery policy specified.	A Publisher Server connection has failed to connect and no recovery policy has been specified.
PUSH-000285	Started publisher '{}'.	A publisher has been started.
PUSH-000286	Starting publisher '{}'.	A publisher is starting.
PUSH-000287	Failed to start Statistics Handler for Publisher '{}'.	A failure has occurred while trying to start the statistics handler for the specified Publisher.
PUSH-000288	Stopped Publisher '{}'.	The given Publisher has been stopped, and could be restarted.
PUSH-000289	Stopping Diffusion Server due to stop-server-if-not-loaded flag for publisher '{}'.	Diffusion failed to start a publisher which has the "stop-server-if-not-loaded" configuration value set to "true" in Publishers.xml, so Diffusion will now be stopped.
PUSH-000290	Failure fetching state of {} from Publisher {} {}.	While handling a fetch request for a topic with no topic data, an error occurred while generating a topic load message.
PUSH-000291	Publisher Server Connection '{}' message queue limit reached.	The specified Publisher Server connection is being closed because the number of messages in its outbound queue has reached the configured limit.
PUSH-000292	Publisher Server Connection {} encountered an unexpected error while sending - closing.	A Publisher Server connection encountered an unexpected error while sending and will be closed.
PUSH-000293	Publisher '{}' unable to connect to Server '{}' - will retry every {} milliseconds.	A Publisher Server connection could not be established but the recovery policy indicates that it should retry and so a an automatic retry will occur at the interval specified.

Id	Message	Description
PUSH-000294	Failed to read dar file '{}'.	Diffusion tried to deploy a publisher from a DAR file, but the DAR was unreadable. Check file permissions and for corruption with the "jar" tool.
PUSH-000295	Publisher '{}' not currently deployed, cannot undeploy.	An attempt was made to undeploy a publisher which wasn't deployed. Either check that this is the case, or that you have the correct spelling of the publisher's name.
PUSH-000296	DeploymentMonitor: One or more of the publishers in '{}' cannot be stopped. Not undeploying file. The dar file will be ignored from now on.	An attempt was made to undeploy a hot-deployed publisher, but that publisher does not return "true" from its isStoppable() method.
PUSH-000297	Output buffer too small ({}). Automatically fragmenting a message to fit; please consider doing this programmatically.	The message being written is larger than the maximum-message-size, and is fragmented to overcome this. This may incur performance penalties.
PUSH-000298	No queue definitions configured : Adding DefaultQueue.	No queue definitions have been configured therefore a default queue configuration is being assumed.
PUSH-000299	A Conflation merge error has occurred in '{}'.	An instance of MessageMerger has thrown an exception from its merge method. The exception is logged and no conflation occurs.
PUSH-000300	Message of size {} is too large for {} which has a limit of {}.	An attempt has been made to send a message to a client that is larger than it is possible to transmit to that client, possibly because of buffer limitations.
PUSH-000301	Default queue definition {} does not exist. '{}' used.	There is no configured queue definition with the name as specified as the default queue therefore the first configured queue is assumed to be the default.
PUSH-000302	Default queue definition not configured. '{}' used.	No default queue definition has been specified in the configuration therefore the first configured queue definition has been assumed to be the default.
PUSH-000303	Queue definition '{}' not known - using default.	The specified queue definition is not known within the configuration therefore the default queue definition has been assumed.
PUSH-000306	Deployment Service failed to process HTTP request.	The publisher deployment service failed to return an HTTP response to the deploying client.
PUSH-000307	PHP Service failed to send 404 response.	An error occurred in the PHP Service while sending a HTTP 404 response to the web client.

Id	Message	Description
PUSH-000308	Failed to process file.	The PHP service was unable to process a PHP file.
PUSH-000309	REST Failed to process service request.	REST Service failed to process a request.
PUSH-000310	Method '{}' from Diffusion Servlet called, not being processed.	The Diffusion Servlet's implementation of the given method does nothing .
PUSH-000311	Diffusion Servlet: {}.	Indicates the startup status of the Diffusion servlet.
PUSH-000312	Client statistics log name '{}' is not configured.	The log named in the Client statistics configuration does not exist in the Logs configuration .
PUSH-000313	Client statistics enabled but no log name supplied.	No log name has been specified in the Client statistics configuration.
PUSH-000316	Failed to update statistics file '{}'.	It was not possible to write the given file.
PUSH-000317	Statistics Manager Failure.	An unexpected error occurred while starting the Statistics Manager.
PUSH-000318	Registering Statistics Reporter '{}'.	A configured statistics reporter of the given name is about to be registered.
PUSH-000319	Failed to register Statistics Reporter '{}'.	An attempt to register the named configured statistics reporter has failed - the reporter has therefore been ignored.
PUSH-000320	Started Statistics Reporter '{}'.	The named Statistics Reporter task has started.
PUSH-000321	Statistics Service failed to load statistics reporter '{}'.	A failure has occurred while attempting to load a configured statistics reporter.
PUSH-000322	Error starting Statistics Service.	An error has occurred while starting the Statistics Service.
PUSH-000323	Background thread pool caught exception executing '{}'.	A task being executed by the Background thread pool has thrown an exception.
PUSH-000324	Thread Pool '{}' could not create new thread.	A Thread Pool encountered an error while attempting to create a new thread.
PUSH-000325	Thread Pool '{}' could not create new thread: {}.	The specified thread pool failed to create an additional thread.
PUSH-000326	Thread Pool {} execution of {} failed.	Failure executing a task in the specified thread pool.
PUSH-000327	Thread Pool {} lower limit notification failed: {}.	The specified thread pool was unable to schedule the notification of the lower threshold limit being reached.

Id	Message	Description
PUSH-000328	Thread Pool {} notification of {} to {} failed.	The execution of a notification to a ThreadPoolNotificationHandler failed.
PUSH-000329	Thread Pool Notification Handler {} loaded for Thread Pool {} with upper threshold of {} and lower threshold of {}.	The specified notification handler has been successfully loaded for the given thread pool.
PUSH-000331	Thread Pool '{}' queue full - aborting execution of '{}'.	Specified thread pool queue has become full and the rejection policy is to abort the task. Either the pool maximum size or the queue size should be increased.
PUSH-000332	Thread Pool {} queue full - task running in current thread.	Specified thread pool queue has become full and the rejection policy is to run the task in the current thread, which may be inefficient. Either the pool maximum size or the queue size should be increased.
PUSH-000333	Thread Pool {} was unable to schedule the execution of notification of rejected execution : {}.	The specified thread pool failed to schedule the notification of a rejected task.
PUSH-000334	Thread Pool {} upper threshold notification failed: {}.	The specified thread pool failed to schedule notification of upper threshold reached.
PUSH-000335	Uncaught Exception in thread '{}'.	An uncaught exception has been thrown from a specified thread and logged.
PUSH-000336	Algorithm '{}' not found.	The given cryptographic module could not be found.
PUSH-000337	File Service: [{}] adding virtual hosts {} and mapping to {}.	A virtual host has been added to the webserver's file service.
PUSH-000338	Unable to process alias entry in file '{}': {}.	Diffusion tried to add a new alias to the webserver, but failed.
PUSH-000339	HTMLTopicDataSegment: unable to process topic data for {}.	The webserver tried to include the contents of a Topic's TopicData into the response, but a failure occurred.
PUSH-000340	HTMLIncludeSegment: unable to populate buffer [{}].	The Web Server tried to include the contents of another file in the response, but a failure occurred writing to a buffer.
PUSH-000341	Missing attribute '{}' from DiffusionTag {} [{}].	The given DiffusionTag incorrect omits the given attribute.
PUSH-000342	Topic '{}' has '{}' TopicData instead of Single Value TopicData. DiffusionTag (TopicData) [{}].	The given topic lacks SingleValueTopicData, making it impossible to insert here.
PUSH-000343	Topic {} does not have topic data. DiffusionTag (TopicData) [{}].	The given topic lacks TopicData and therefore any state, making it impossible to insert here.

Id	Message	Description
PUSH-000344	Can't find file attribute in DiffusionTag [{}].	The given DiffusionTag incorrectly omits the 'file' attribute.
PUSH-000345	Cannot load include file '{}' from DiffusionInclude '{}': {}.	An error occurred loading the given file.
PUSH-000346	Cannot read specified file for DiffusionTag '{}' [{}].	An error occurred loading the given file specified in the DiffusionTag.
PUSH-000347	Unknown publisher in DiffusionTag {} [{}].	The Publisher named in this DiffusionTag is not loaded, or does not exist.
PUSH-000348	Unknown topic for DiffusionTag (TopicData) {} [{}].	The Topic named in this DiffusionTag does not exist.
PUSH-000349	Unknown type attribute '{}' for DiffusionTag [{}].	The type attribute of this DiffusionTag holds an unexpected value.
PUSH-000350	Web Server service cannot process HTML tag '{}' for the publisher '{}'.	An error occurred while processing an HTML embedded tag.
PUSH-000351	{}: Connection rejected as request was missing the HTTP header field '{}'.	This connection was rejected as it was missing the given HTTP header. The connection is likely not from a Diffusion client, or an intermediary (firewall/load-balancer) has removed it.
PUSH-000352	{}: connection rejected as [{}] did not match '{}'.	The connection from an untrusted host was rejected, in accordance with CORS configuration.
PUSH-000353	{}: Connection rejected as [{}] did not match {}.	The connection was rejected because it's websocket origin did not match the regular expression stored in the configuration.
PUSH-000354	{}: cors-origin not defined or invalid. Aborting request.	CORS (Cross Origin Resource Sharing) request has been received by the Web Server Client Service but no "cors-origin" has been configured or the configured value was invalid. The request has been aborted.
PUSH-000355	{} unable to process HTTP Request '{}'.	Web Server File Service was Unable to process the given HTTP Request.
PUSH-000356	{}: HTTP Processing error on connector '{}': {}.	An error occurred handling an HTTP request.
PUSH-000357	{}: Invalid regular expression '{}' for CORS origin '{}' - feature disabled.	An invalid regular expression was given for the CORS origin configuration.
PUSH-000358	{}: Invalid regular expression '{}' for WebSocket origin '{}' - feature disabled.	An invalid regular expression was given for the WebSocket origin configuration.
PUSH-000359	Poll request for client '{}' from '{}' failed as no matching client can be found. Client may have been	The Diffusion Web Server has received a poll request for a Client that is not known to the server. This may be because the Client has been

Id	Message	Description
	disconnected previously. Poll request will be ignored.	closed by the Server. This may also indicate an incorrectly configured load balancer, routing an HTTP connection to a server that was not servicing the given client. Consider enabling session-stickiness for HTTP Diffusion clients.
PUSH-000360	FileService [{}]: not adding virtual hosts '{}' mapping to '{}' as it is not a directory.	Failed to add a virtual host to the webserver's file service because the configured mapping does not point to a directory.
PUSH-000361	Unable to service HTTP request for service '{}'.	An HTTP service was invoked, but it failed to process the incoming request.
PUSH-000362	Started HTTP Service for '{}'.	An HTTP service was successfully started.
PUSH-000363	{} Unable to create log.	An HTTP service has its own log file defined, but Diffusion was unable to create the logger.
PUSH-000364	Web Server {} failed to instantiate HTTP service '{}' : {}.	A web server failed to instantiate the named configured HTTP service.
PUSH-000365	Web Server failed to start cache service for virtual host '{}'.	An error occurred while starting the cache service for the given virtual host.
PUSH-000366	Virtual Host HTTP request {} is invalid.	An HTTP request being handled by a virtual host is trying to break out of the virtual root.
PUSH-000367	Virtual Host HTTP request {} - unable to read file {}.	A failure has occurred trying to read the specified HTML file.
PUSH-000368	{}: websocket-origin not defined or invalid. Aborting request.	Web Server Client Service has received a Web Socket request but "websocket-origin" has not been configured or is invalid. The request has been aborted.
PUSH-000369	Failed to schedule a periodic tidy of the WhoIs cache.	A failure has occurred while trying to schedule a periodic tidy of the WhoIs cache.
PUSH-000370	WhoIs Provider class instantiation failure calling '{}'.	Unable to instantiate the configured WhoIs provider. Will use the default WhoIs provider.
PUSH-000372	Unable to load GeoIP Database '{}'.	An error occurred while loading or initialising the GeoIP database.
PUSH-000373	WhoIs Service failed to lookup address '{}'.	A failure has occurred in the WhoIs service when trying to look up the given address.
PUSH-000374	WhoIs service starting with {} thread(s).	The WhoIs service is starting.

Id	Message	Description
PUSH-000375	Deprecated connector type RTMP found in Connectors.xml and will be ignored.	Deprecated connector type RTMP found in Connectors.xml and will be ignored.
PUSH-000376	Deprecated element '{}' found in {}.xml.	A deprecated element has been found in specified properties file and should be removed.
PUSH-000377	Failed to load XML properties '{}'.	Failed to load XML properties.
PUSH-000378	Failed to load publisher '{}' configuration from XML properties.	Diffusion was unable to load the configuration for the named publisher from an XML properties file.
PUSH-000379	Failed to load subscription validation policy '{}' from file '{}' : {}.	A failure has occurred loading an XML subscription validation policy file.
PUSH-000380	Failed to parse {} at line {}, column {}: {}.	An XML validation event has occurred while validating an XML property file.
PUSH-000381	Loaded XML {} properties from '{}'.	The specified XML properties have been loaded from the file indicated.
PUSH-000382	Unable to load Publisher config file {}.	Attempted to load a publisher's configuration file (Publishers.xml), but it couldn't be found or is unreadable.
PUSH-000383	unable to check server-side addressing for product {} [{}] : {}.	An error occurred while attempting to check server side addressing in the licence file.
PUSH-000384	'{}': Failed to decode licence key '{}'.	Unable to decode licence key.
PUSH-000385	Failed to load subscription validation topic mapping for Publisher {} topic {} from file {} : {}.	A failure has occurred loading a publisher topic mapping for a subscription validation policy.
PUSH-000386	Failed to load connection validation policy {} from {}.	A failure has occurred loading XML Connection Validation Policy from the specified file.
PUSH-000387	Licence check failed to match MAC address for '{}'.	Unable to match a MAC address for the specified product.
PUSH-000388	Failed to save XML Properties '{}'.	A failure has occurred while saving XML properties.
PUSH-000389	Server Connection '{}' has been added for Publisher '{}'.	A configured server connection has been successfully added for a Publisher.
PUSH-000390	Failed to add Server Connection '{}' for Publisher '{}'.	SSL keystore file has successfully been loaded for a Publisher to server connection.
PUSH-000391	Server Connection '{}' for Publisher '{}' loaded SSL key store '{}'.	A publisher has created a server connection that has loaded a key store to establish the SSL context. The key store is used to securely store key, certificate and trust information.
PUSH-000392	Publisher '{}' has lost connection to Server '{}' - Publisher closing.	A Publisher has lost connection to a server and the recovery policy

Id	Message	Description
		indicates that the publisher should be closed.
PUSH-000393	Publisher '{}' has lost connection to Server '{}' - no recovery policy specified.	A Publisher has lost connection to a server and no recovery policy has been specified.
PUSH-000394	Publisher '{}' has lost connection to Server '{}' - will retry every {} milliseconds.	A Publisher has lost connection to a server and the recovery policy is to retry therefore a retry will occur at the specified interval.
PUSH-000395	Publisher '{}' unable to schedule retry connection to '{}' - publisher stopping.	A failure has occurred attempting to schedule a connection retry between a Publisher and a server. The publisher will be closed.
PUSH-000396	Unable to perform substitution within Property Value '{}' due to syntax error.	A syntax error has been detected whilst trying to perform environment variable substitution on a configuration property.
PUSH-000397	Deploying publishers in dar file '{}'.	Deploying the publishers in a single DAR file. There may be multiple publishers in a single DAR file. The publishers will be extracted from the DAR file and the additional configuration files loaded.
PUSH-000399	Directory '{}' does not exist or is not a directory. Using '{}' as a log directory.	The directory of a log definition or the default log directory does not exist or it is not a directory. In this case the temporary file directory is used instead. This is specified by 'java.io.tmpdir'. If the intended directory does exist the problem may be that the relative path is incorrect because the working directory is not what you expect. The simplest way to resolve this is to use absolute file paths to reference log directories. This may be shown when the log is created and when checking if it needs to be rotated.
PUSH-000400	Unable to add listener when reconnecting to {}.	While reconnecting, the JMS adapter was unable to add a listener for a JMS Destination.
PUSH-000401	Unable to create consumer when reconnecting to {}.	While reconnecting, the JMS adapter was unable to recreate a JMS Consumer.
PUSH-000402	Unable to create session when reconnecting to {}.	While reconnecting and re-establishing the existing JMS Consumers, the JMS adapter failed to create a JMS Session.
PUSH-000403	'{}': Connection has been closed with an open request.	The TCP connection for an HTTP request that had not been responded to has been closed. This may indicate a network problem with the client.

Id	Message	Description
		Alternatively it may be caused by the closing of a browser, the close message may be sent to the server after the poll connection has been lost. The client will be closed to indicate that there was an IO problem or that the connection was lost.
PUSH-000404	'{}': No poll has been opened by client within poll timeout.	HTTP based connections require an open polling connection to receive messages. If the client has no open polling connection for a fixed period the client will be closed as unresponsive.
PUSH-000405	Message not sent. The Event Publisher '{}' may not be connected.	The Event Publisher attempted to send with a null Multiplexer. This may be due to the Event Publisher not being connected.
PUSH-000406	Application handler threw exception {}. [cid={}].	An application handler threw an exception when called.
PUSH-000407	An unexpected response was received from a peer and ignored. [cid={} response={}].	A response for a different conversation set has been received from a peer and will be ignored. Typically this indicates a response has been delivered late, after communication with the peer has been re-established.
PUSH-000408	A response was received from a peer for a finished conversation and ignored. [cid={} response={}].	A response with an unknown conversation ID has been received from a peer. Typically this indicates the conversation has timed out.
PUSH-000409	Unable to close socket channel '{}'.	The socket channel failed to close.
PUSH-000410	Failed to queue background close of the Publisher Server Connection '{}'.	An attempt to queue the close of the Publisher Server Connection failed, therefore the connection has been closed in the current thread.
PUSH-000411	Multiplexer undefined. The PublisherServerConnection '{}' may not be connected.	The Publisher Server Connection attempted to send with a null Multiplexer. This may be due to the PublisherServerConnection not being connected.
PUSH-000412	Executor {}: task '{}' finished abnormally.	A background task has thrown an exception. See the log for details.
PUSH-000413	Received request from control client '{}' for the details of client session '{}'.	The server has received a control client request for details of a client session.
PUSH-000414	Received request from control client '{}' for the details of client session '{}'. The client session does not exist.	The server has received a control client request for details of an unknown client session.

Id	Message	Description
PUSH-000415	Received request from control client '{}' to close client '{}' for reason '{}'.	The server has received a request from a control client to close a client session.
PUSH-000416	Received request from control client '{}' to close client '{}' for reason '{}'. The client session does not exist.	The server has received a request from a control client to close an unknown client session.
PUSH-000417	Fetch operation {} for {} was aborted due to missing responses for the following topics: {}.	A fetch operation failed, possibly because of a publisher or control client issue. Look for earlier messages in the server log.
PUSH-000418	Control provided fetch content for topic '{}' and client '{}', but this server failed to create a fetch response message.	A control client provided a response for a topic fetch request from the client, but it could not be converted to a message.
PUSH-000419	One or more control clients have registered to provide a fetch response for topic '{}', but none was available for request from client '{}'.	No control client was available to handle the request, or one was chosen and failed while processing. The client fetch handler will be discarded.
PUSH-000420	One or more control clients have registered to handle queue events, but none was available for a {} event for client '{}'.	No control client was available to handle the queue event, or one was chosen and failed while processing.
PUSH-000421	Received request from control client '{}' to set conflation for client '{}' to '{}'.	The server has received a request from a control client to change the conflation setting for a client queue.
PUSH-000422	Received request from control client '{}' to set conflation for client '{}' to '{}'. The client session does not exist.	The server has received a request from a control client to change the conflation setting for an unknown client session.
PUSH-000423	Received request from control client '{}' to throttle message queue for client '{}' using the {} policy, limit={}.	The server has received a request from a control client to change the throttling policy of a client session.
PUSH-000424	Received request from control client '{}' to throttle message queue for client '{}' using the {} policy, limit={}. The client session does not exist.	The server has received a request from a control client to change the throttling policy of an unknown client session.
PUSH-000426	{}: command service call failed, reporting error to {}: {}.	An internal service call has failed. An error will be reported to the client session listener.
PUSH-000427	Attempt by a Classic API client ({}) to subscribe to the service topic: '{}'. This is reserved for Unified API clients.	Unified API clients use an internal service topic to communicate with the server. Classic API clients may not subscribe to it. Attempts to subscribe to it will be ignored by Diffusion and the attempt logged.
PUSH-000428	'{}': Has rejected an attempt to connect from a Unified API client. Unified API clients are not supported by this connector.	The connector has been configured to accept connections only of certain client types. An attempt was made to connect by a client

Id	Message	Description
		of an unsupported type. Either the connector is configured for the wrong type, the client should have connected to a different connector or an attempt was made to connect a client that has greater functionality than you allow on this connector.
PUSH-000429	'{}': Has rejected an attempt to connect from a Classic API client. Classic API clients are not supported by this connector.	The connector has been configured to accept connections only of certain client types. An attempt was made to connect by a client of an unsupported type. Either the connector is configured for the wrong type or the client should have connected to a different connector.
PUSH-000430	There are currently no controllers for {}.	A control binding has been established, but there are no available controllers.
PUSH-000431	Rejected attempt by client {} to bind a control handler: {}.	This may indicate that another control group has established a binding, or that the client already has a binding.
PUSH-000432	Attempt by client {} to register or unregister from an unknown control point: {}.	This is unexpected, and possibly indicates the server and client versions are incompatible.
PUSH-000433	Application handler threw exception {} when called with a response. [cid={} response={}].	An application handler threw an exception when called.
PUSH-000434	Diffusion finished deploying {} components.	Diffusion deployments completed.
PUSH-000435	Task failure in selector thread '{}'.	A task failed to be executed in a selector thread.
PUSH-000436	Unable to create file handler for logger '{}'. Log messages will still appear in the console. : {}.	A file handler was unable to be created for the requested logger. Messages will continue to be logged to the console, but will not be recorded to their own file.
PUSH-000437	A message has been received with an unknown topic alias '{}'. It cannot be parsed.	When topic aliasing is enabled a short alias is used to replace topic names. This reduces the size of messages sent. A client still needs to know the topic name so the topic name and alias are sent together in the topic load. If a client does not receive a topic load it is unable to handle the deltas with only the alias. Diffusion assumes that a client will receive a topic load message if a load message is created for that topic. If the message is not sent then this error message will be logged.

Id	Message	Description
PUSH-000439	The authentication manager failed to start because an authentication handler of class {} could not be created.	An instance of a configured authentication handler could not be created.
PUSH-000440	Authentication handler {} threw an exception, denying authentication to {}.	A configured authentication handler failed to process an authentication request.
PUSH-000441	The authorisation manager failed to start because an authorisation handler of class {} could not be created.	An instance of a configured authorisation handler could not be created.
PUSH-000442	TopicFetchHandler for topic '{}' returned null, indicating an asynchronous response. This is not supported by the client '{}'. The fetch response (if any) will be discarded.	The client does not support asynchronous fetch responses from a TopicFetchHandler.
PUSH-000443	Publisher '{}' returned null for fetch of topic '{}', indicating an asynchronous response. This is not supported by the client '{}'. The fetch response (if any) will be discarded.	The client does not support asynchronous fetch responses from a publisher.
PUSH-000444	Unable to read aliases file '{}'.	The configured aliases file could not be read.
PUSH-000445	Discarding WhoIs request for {} due to backlog. There are currently {} pending requests.	A WhoIs request has been discarded because there is currently a large number of pending requests. The WhoIsListener will be notified with basic locale information based on the callers IP.
PUSH-000446	Unable to read subscription polices from '{}'.	The configured subscription policy file could not be read.
PUSH-000447	Failed to load publisher '{}'.	Diffusion was unable to register a publisher, most likely due to a problem with the publisher's definition.
PUSH-000448	An error occurred registering a handler with parameters '{}': {}.	An internal error occurred registering a control client handler.
PUSH-000450	An error occurred unregistering a handler with parameters '{}': {}.	An internal error occurred unregistering a control client handler.
PUSH-000452	A topic source update failed because no updater could be found.	A topic source update failed because the client has unregistered the updater.
PUSH-000453	A topic source update failed unexpectedly: {}.	A topic source update failed. See the log message for further information.
PUSH-000454	Server requested unsupported service {}.	The server has requested an internal service that the client has not registered.

Id	Message	Description
PUSH-000455	Client requested unsupported service {}.	The client has requested an internal service that the server does not support.
PUSH-000456	Notifying session {} of error '{}'.	A session error has occurred. The session error handler will be called.
PUSH-000457	More multiplexers configured than available CPU cores ({} vs {}).	Assigning more multiplexers than there are available CPU cores may lead to degraded performance.
PUSH-000458	Application classpath entry '{}' in is not a directory. Check user libraries in the server configuration.	The names in the user library configuration should identify directories.
PUSH-000459	The configured auto-deployment directory '{}' does not exist.	The configured auto-deployment directory '{}' does not exist.
PUSH-000460	Class loader failed to enumerate files in directory '{}'.	An error occurred reading the files within the given directory.
PUSH-000461	Unable to add file '{}' to the class loader.	A server class loader failed to load the given file.
PUSH-000462	Multiplexer '{}' overflowed before start up - event discarded.	An event-driven multiplexer before it was started.
PUSH-000463	Authorisation manager handler has changed from '{}' to '{}'.	The Authorisation Manager has changed the internal Authorisation Handler to a new handler instance.
PUSH-000464	Update failed for topic '{}'.	An error was encountered whilst attempting to update a topic.
PUSH-000465	Connector '{}' has pings disabled. Unresponsive HTTP clients will not be detected.	The server relies on pinging in order to detect unresponsive HTTP clients. Unresponsive clients may go unnoticed if the ping frequency is not set.
PUSH-000466	Message stream '{}' threw exception.	A message stream callback raised an exception when called.
PUSH-000467	Unable to establish a tunnel through the proxy at {}:{}.	There was a problem in establishing a tunnel through the specified proxy. There may be an issue in reading from or writing to the proxy.
PUSH-000468	Topic stream '{}' threw exception.	A topic stream callback raised an exception when called.

Chapter 20

Introspector

In this section:

- Supported platforms.
- Installing from update site
- Installing subsequent plugin updates.
- Uninstalling
- Opening the Diffusion perspective
- Adding servers
- Opening servers
- Exploring the topics
- Getting topic values
- Configuring columns
- Ping servers
- Count topics
- Using the clients view
- Ping
- Statistics
- Topics
- Logging
- Server logs
- Property obfuscator

An introduction to the Introspector Eclipse plugin

This section introduces and explains the features of the Diffusion Introspector GUI. This is currently a beta-product, and as such might contain bugs.

The Introspector GUI is a set of Eclipse plugins. This product depends on a minimum of Eclipse version Juno or later for Java EE Developers which can be downloaded from http://eclipse.org/downloads/.

This GUI uses features from the existing system publisher as well as the new Diffusion publisher for which a license must be obtained.

Supported platforms.

The stack of technologies upon which this software is expected to function.

Introspector is tested on version Juno or later for Java EE Developers and Java HotSpot Development Kit 7 (update 67).

Installing from update site

All Diffusion Eclipse plugins are distributed primarily through the Eclipse Update site.

The Eclipse Update site is located at the following URL: http://download.pushtechnology.com/eclipse/5.1.

A To install the software, ensure you have version Juno or later for Java EE Developers installed. Earlier versions might work, but are not supported. Pick the **Help** menu, click the **Install new software** menu item. From the dialog, choose the **Add ...** button. Complete the dialog as shown below.

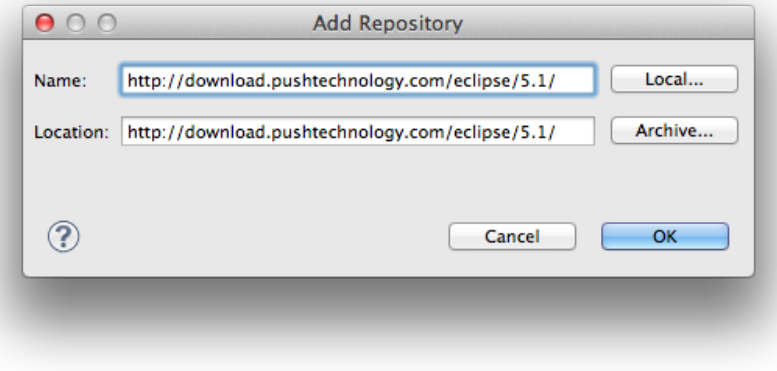


Figure 71: Adding a repository

Click on the **OK** button to go to the Available Software dialog.

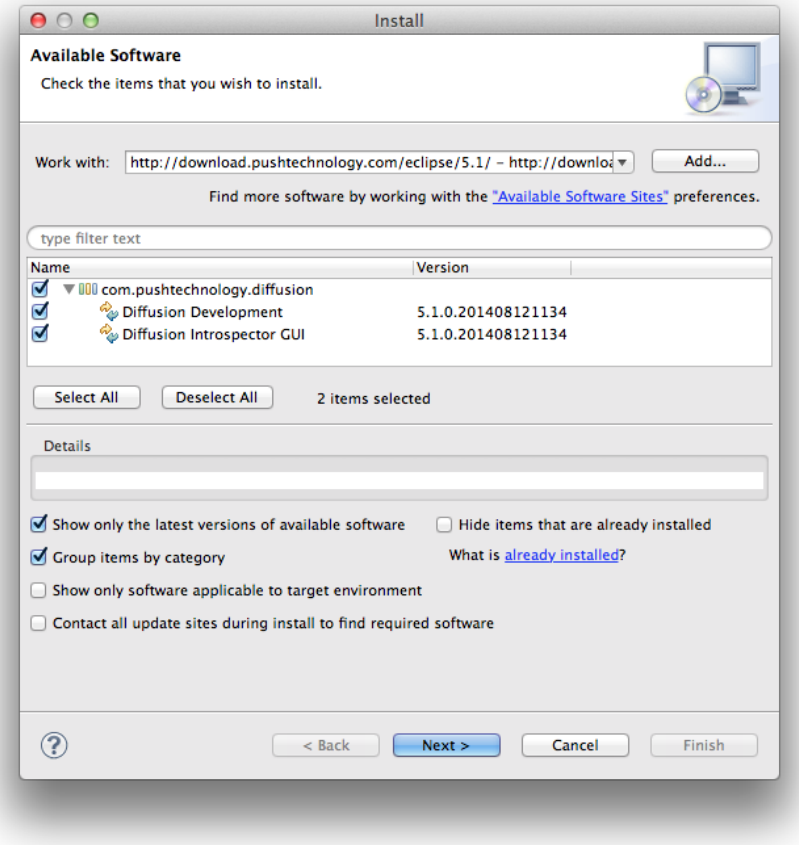


Figure 72: Install dialog

Click the **Next** button. Follow the sequential stream of dialogs. When you are given the dialog shown below, select **I accept the terms of the license agreement** if you agree to the license terms.

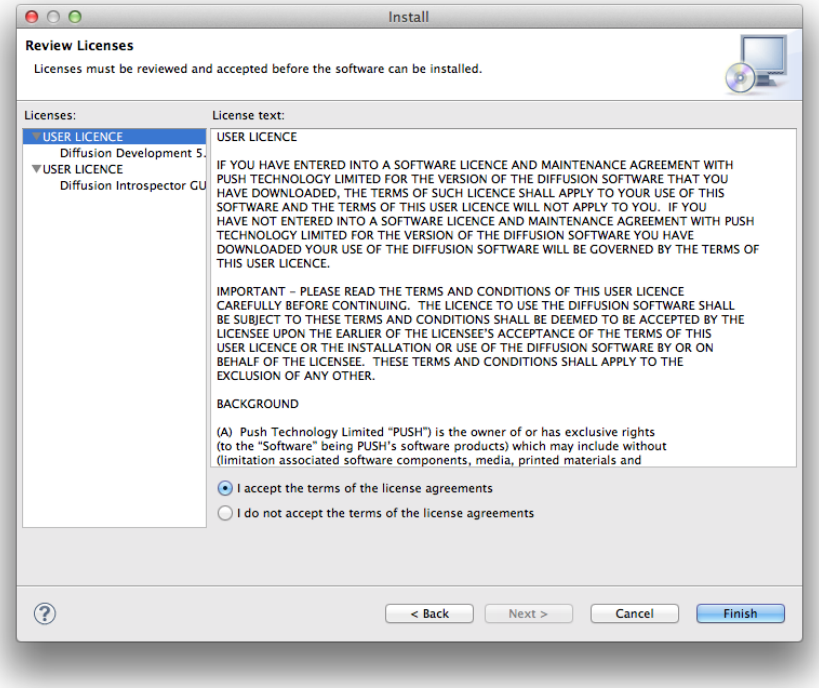

Install
Review Licenses
Licenses must be reviewed and accepted before the software can be installed.
Licenses:
License text:
USER LICENCE
Diffusion Development 5.
USER LICENCE
Diffusion Introspector GU
USER LICENCE
IF YOU HAVE ENTERED INTO A SOFTWARE LICENCE AND MAINTENANCE AGREEMENT WITH PUSH TECHNOLOGY LIMITED FOR THE VERSION OF THE DIFFUSION SOFTWARE THAT YOU HAVE DOWNLOADED, THE TERMS OF SUCH LICENCE SHALL APPLY TO YOUR USE OF THIS SOFTWARE AND THE TERMS OF THIS USER LICENCE WILL NOT APPLY TO YOU. IF YOU HAVE NOT ENTERED INTO A SOFTWARE LICENCE AND MAINTENANCE AGREEMENT WITH PUSH TECHNOLOGY LIMITED FOR THE VERSION OF THE DIFFUSION SOFTWARE YOU HAVE DOWNLOADED YOUR USE OF THE DIFFUSION SOFTWARE WILL BE GOVERNED BY THE TERMS OF THIS USER LICENCE.
IMPORTANT – PLEASE READ THE TERMS AND CONDITIONS OF THIS USER LICENCE CAREFULLY BEFORE CONTINUING. THE LICENCE TO USE THE DIFFUSION SOFTWARE SHALL BE SUBJECT TO THESE TERMS AND CONDITIONS SHALL BE DEEMED TO BE ACCEPTED BY THE LICENSEE UPON THE EARLIER OF THE LICENSEE'S ACCEPTANCE OF THE TERMS OF THIS USER LICENCE OR THE INSTALLATION OR USE OF THE DIFFUSION SOFTWARE BY OR ON BEHALF OF THE LICENSEE. THESE TERMS AND CONDITIONS SHALL APPLY TO THE EXCLUSION OF ANY OTHER.
BACKGROUND
(A) Push Technology Limited "PUSH") is the owner of or has exclusive rights (to the "Software" being PUSH's software products) which may include without (limitation associated software components, media, printed materials and
I accept the terms of the license agreements
I do not accept the terms of the license agreements
< Back
Next >
Cancel
Finish

Figure 73: Accept the license agreement

If asked to select the certificate to trust, select **Push Technology Ltd; Development; Push Technology Ltd** and click **OK**.

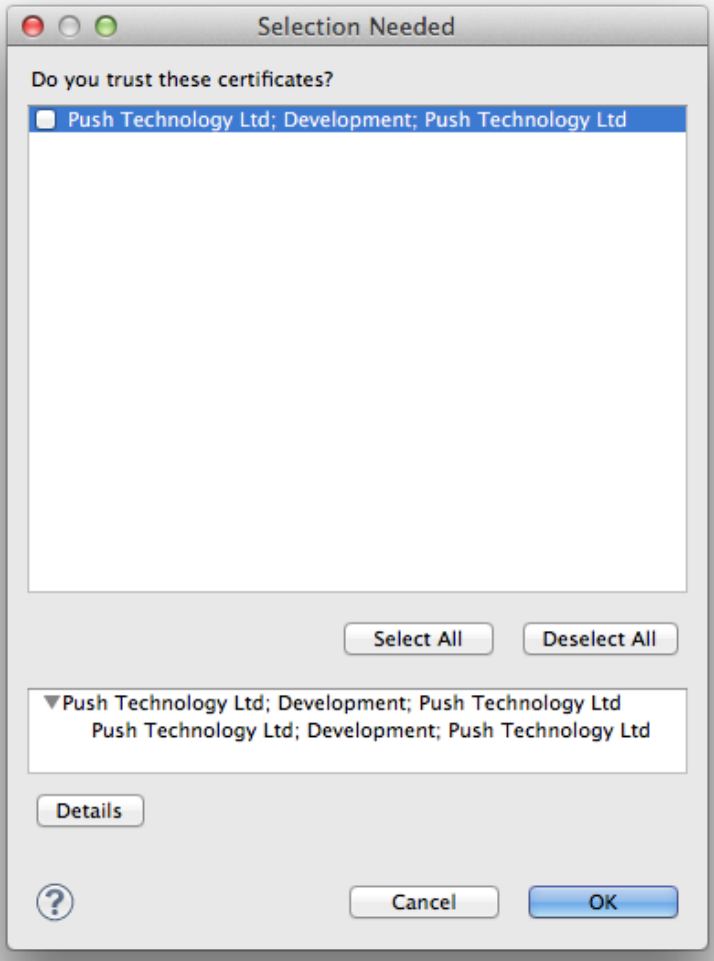


Figure 74: Click OK

When the installation process is done, click the **Restart Now** button.

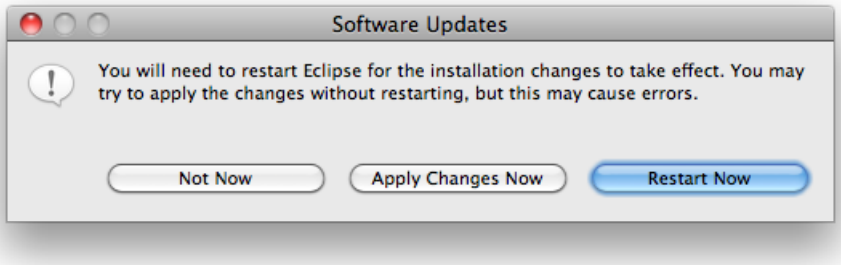

Software Updates
You will need to restart Eclipse for the installation changes to take effect. You may try to apply the changes without restarting, but this may cause errors.
Not Now
Apply Changes Now
Restart Now

Figure 75: Restarting

Installing subsequent plugin updates.

Once this process is complete users can install the latest version of the plugins by clicking on the **Help** menu and picking the **Check for updates** menu item.

Uninstalling

Eclipse maintains a registry of what plugins have been installed, and gives users a means of removing plugins from it.

Firstly, open the **About Eclipse** dialog. On OS X this can be found in the **Eclipse** menu, on every other platform – the 'Help' menu.

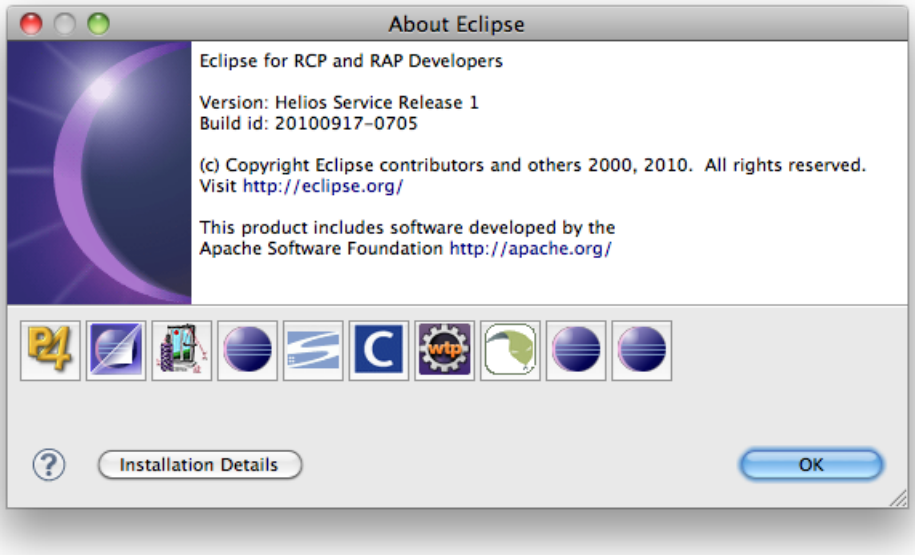

About Eclipse
Eclipse for RCP and RAP Developers
Version: Helios Service Release 1
Build id: 20100917-0705
(c) Copyright Eclipse contributors and others 2000, 2010. All rights reserved.
Visit http://eclipse.org/
This product includes software developed by the
Apache Software Foundation http://apache.org/
Installation Details
OK

Figure 76: About Eclipse dialog

Click on **Installation details**. Once the **Eclipse Installation Details** dialog appears, click on the **Installed Software** tab to review which plugins are installed.

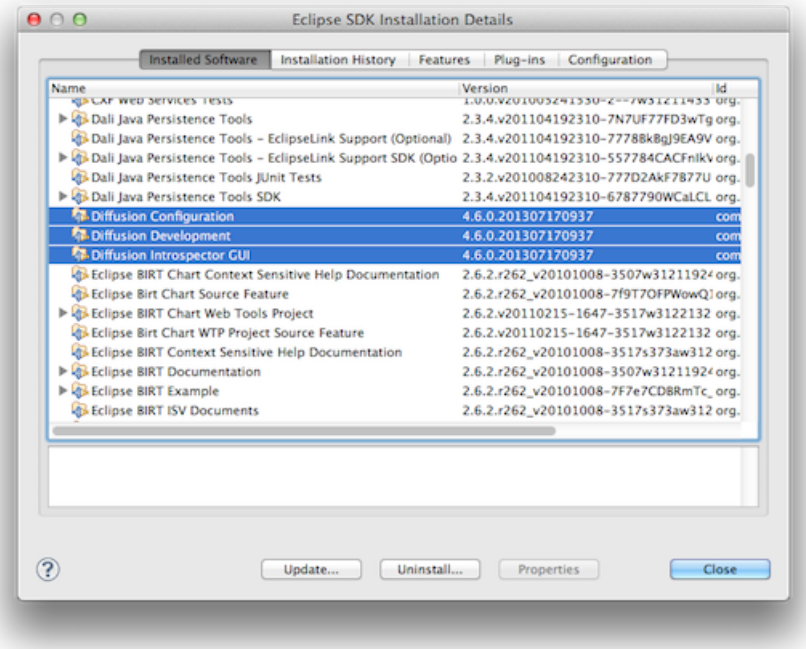


Figure 77: Installed plugins

Once a the software is found, you can uninstall it using the **Uninstall ...** button. Users might be advised to restart Eclipse once the process is complete.

Opening the Diffusion perspective

The Diffusion perspective contains a set of views that relate to Diffusion functional.

Select **Window** > **Open Perspective** > **Other ....** Pick the **Diffusion** item from the list and click on **OK**.

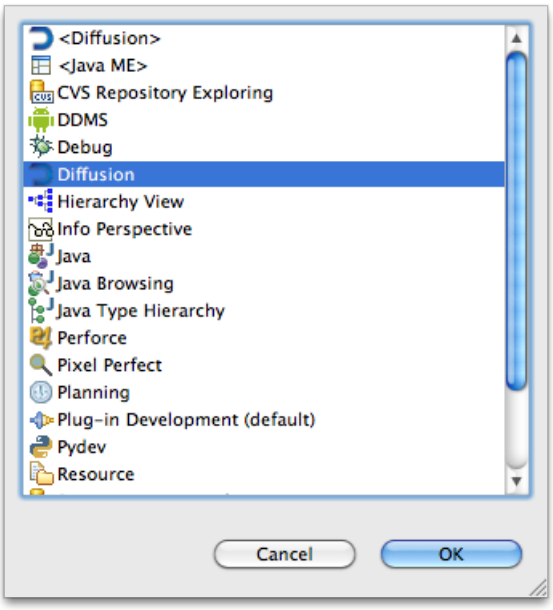


Figure 78: Perspective

Opening the Diffusion perspective presenta to the user a set of views related to Diffusion functionality. The same views can be added to other perspectives (such as the Java perspective):

- Select **Window** > **Show View** > **Other** ....
- Find the Diffusion section in the list and pick the views required.

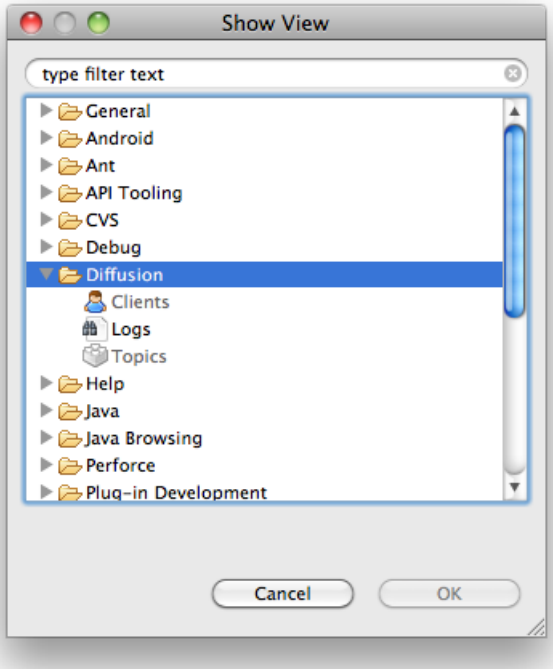


Figure 79: Views

The Diffusion perspective shows the topics and clients views on the left hand side.

Adding servers

To add server details to the plugin click the green plus button in the head of the topic view or client view.

You can click on the **Test** button to test the connection before clicking **OK**.

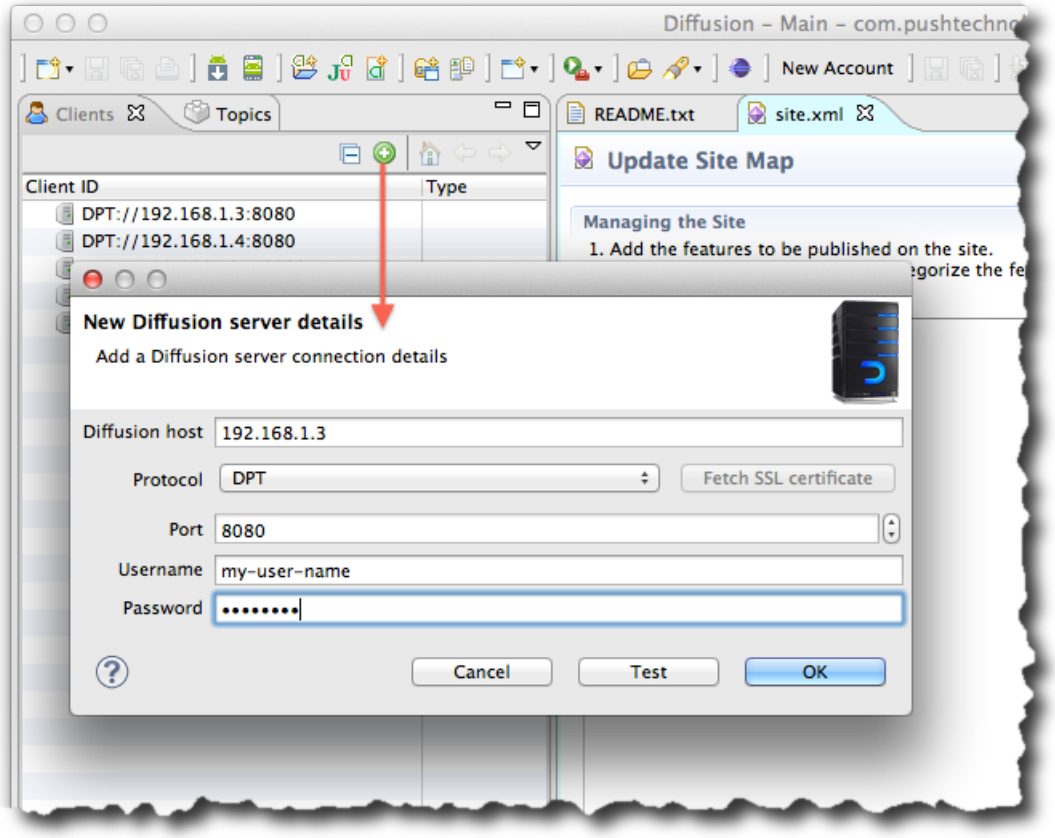


Figure 80: Add a server

If the you want to amend server details, the context menu of closed server (that is, those servers that are not currently connected) holds an **Edit** item that presents the same dialog box.

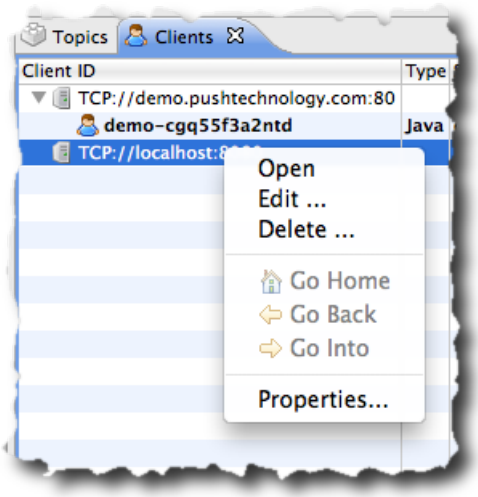


Figure 81: Edit server details

Opening servers

You can view server information through Eclipse.

In either the topics view or the clients view, right-click on a server icon and pick **Open** from the context menu, or double-click on the icon.

The set of columns shown by default show the Diffusion server name (server.name in `etc/Server.xml`) in the value column and the time that the server was started in the **Created** column.

Exploring the topics

You can navigate the tree of topics the same way as any other tree control.

The GUI populates the tree on-demand and asynchronously, so a branch of the tree might be populated some short time after it has been opened. Each topic shows its name, the publisher, the type of any topic-data associated, the date-stamp the topic was created, the date-stamp the topic was last updated and the value of that topic. The last two are populated only if the topic is fetched or subscribed.

The topic tree is live and as new topics are created and old topics deleted so the tree is updated.

Getting topic values

Users can retrieve the current state of the topic by picking **Fetch** from the context menu of a topic. Picking **Fetch recursively** also fetches the values of all topics below this one.

Users can also subscribe to a topic by picking **Subscribe** from the context-menu of a topic. **Subscribe recursively** also subscribes to values of all topics below this one. When the value of a topic changes the **Value** and **Updated** columns flash blue for a second.

To drill into a part of the topic tree they can pick **Go Into** from the topic context menu. The view shows the name of the top-level topic in the view header.

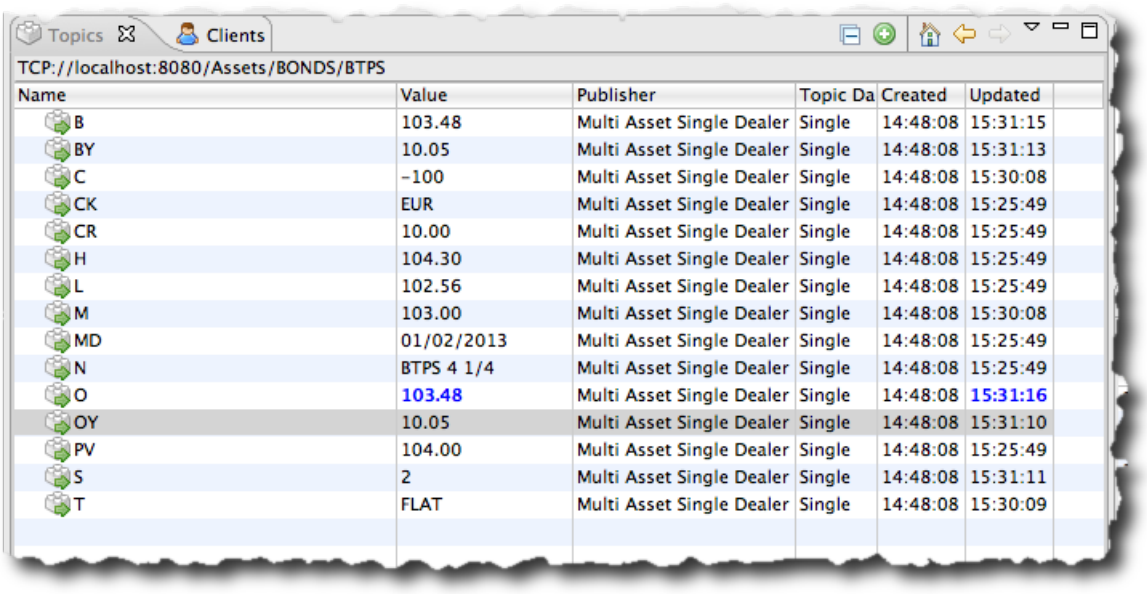

Topics
Clients
TCP://localhost:8080/Assets/BONDS/BTPS
Name
Value
Publisher
Topic Da
Created
Updated
B 103.48 Multi Asset Single Dealer Single 14:48:08 15:31:15
BY 10.05 Multi Asset Single Dealer Single 14:48:08 15:31:13
C -100 Multi Asset Single Dealer Single 14:48:08 15:30:08
CK EUR Multi Asset Single Dealer Single 14:48:08 15:25:49
CR 10.00 Multi Asset Single Dealer Single 14:48:08 15:25:49
H 104.30 Multi Asset Single Dealer Single 14:48:08 15:25:49
L 102.56 Multi Asset Single Dealer Single 14:48:08 15:25:49
M 103.00 Multi Asset Single Dealer Single 14:48:08 15:30:08
MD 01/02/2013 Multi Asset Single Dealer Single 14:48:08 15:25:49
N BTPS 4 1/4 Multi Asset Single Dealer Single 14:48:08 15:25:49
O 103.48 Multi Asset Single Dealer Single 14:48:08 15:31:16
OY 10.05 Multi Asset Single Dealer Single 14:48:08 15:31:10
PV 104.00 Multi Asset Single Dealer Single 14:48:08 15:25:49
S 2 Multi Asset Single Dealer Single 14:48:08 15:31:11
T FLAT Multi Asset Single Dealer Single 14:48:08 15:30:09

Figure 82: View topic values

Click the back button to return to the previous position in the topic tree.

Configuring columns

Columns can be reordered with drag and drop. Column positions are stored between uses. The context menu of the headers lets users toggle individual columns on and off.

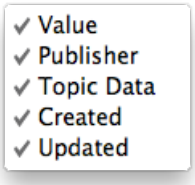


Figure 83: Re-order columns

Ping servers

To establish the response time of a remote server pick **Ping** from the context menu of a connected server. The results in milliseconds is shown next to the server name in the **Name** column.

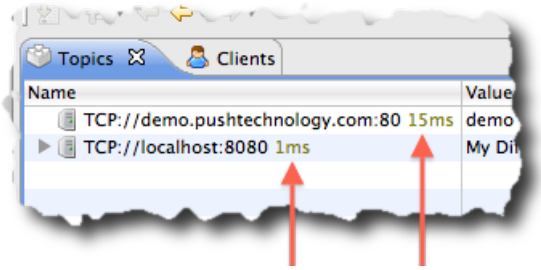


Figure 84: Ping a server

Count topics

Pick **Count topics** from the context-menu of a topic for the server to count the number of topics underneath this one.

The result is shown to the right of the topic name

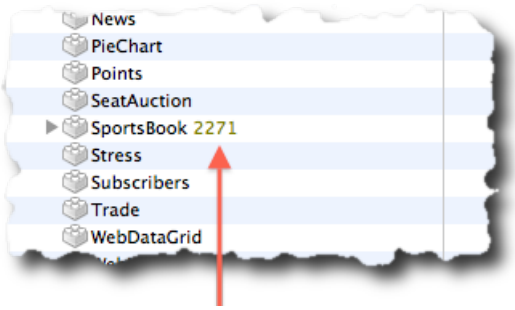


Figure 85: Topic count

Using the clients view

The clients view is a live view of the clients connected to the set of Diffusion servers.

As the Diffusion GUI is also a Diffusion client, it shows up in this list and is shown in bold. A newly connected client is bright blue for 1s, while a disconnecting client is drawn pale gray for 1s before it vanishes.

The properties of a client listed in the following table.

Table 95: Client properties in the Eclipse client view

Type	Flash, Silverlight, WebSockets, iOS, or Android, and so on.
Server	the name of the server this client is connected to
Start time	the time-stamp the client connected to its server
Resolved name	the result of a WHOIS query on the IP address of the remote client
Description	more details from the WHOIS query
Locale	The nation or territory from which this client is connecting
Ping	The time taken to ping the remote client
Alarm	Not currently in use
IP address	The IP address of the remote client
Client reference	The reference/description granted to a client

Ping

Using this feature the user can ping individual clients, the results of which show up in the **Ping** column

All Diffusion clients respond to ping requests from the server.

Further details of a client can be gained from the properties view. Pick **Properties** from the context menu of a client.

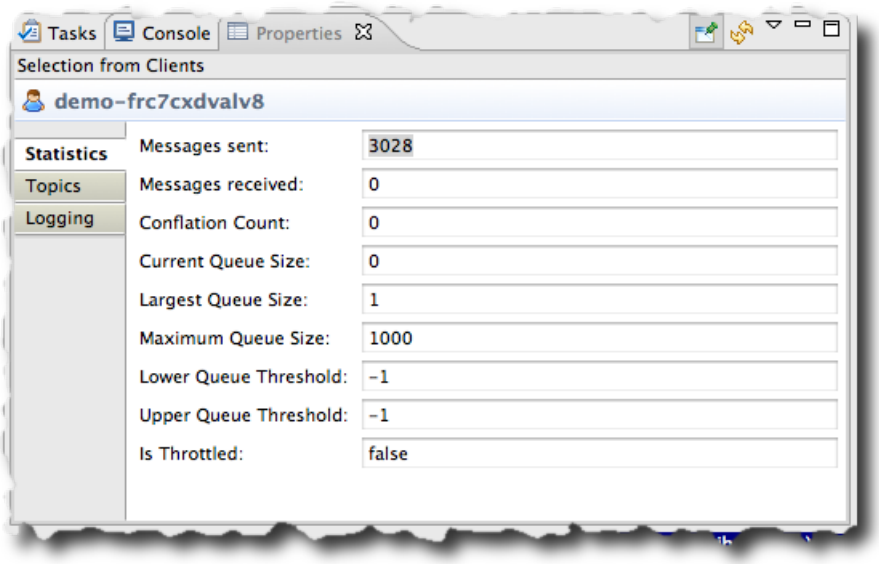


Figure 86: Ping clients

The **Properties** view is part of the Eclipse framework and used in many other non-Diffusion circumstances. Consequently, if the user clicks a new object in another view the **Properties** view might show the properties of that object instead.

Statistics

This shows the number of messages sent, received, the number of messages conflated, the current queue size, the highest queue size, the maximum queue size allowable and throttling thresholds.

Unlike previous views this data is not live, and the refresh button in the view header must be clicked to refresh this view.

Topics

This section shows the topics to which this client is currently subscribed along with any related topic reference. This data is also not live, and must be refreshed manually.

Logging

Access to remote client logs is currently an experimental and optional feature. Those clients that do support it subscribe to the Diffusion/ClientLogs topic.

Using this section of the view, you can retrieve log entries from remote JavaScript clients that are normally sent to the JavaScript console alone.

The JavaScript environment inside a modern browser is memory constrained. This facet of the GUI lets the user set the level at which a log entry is stored and the size of the buffer in which entries are stored before being removed.

Server logs

Diffusion server logs are published through the Diffusion/Logs topic as well as written to log files. The Logs view subscribes to Diffusion to display the log entries as they happen.

Right-click on any open Diffusion server item in the topics or clients view and pick **Logs ...** from the context menu.

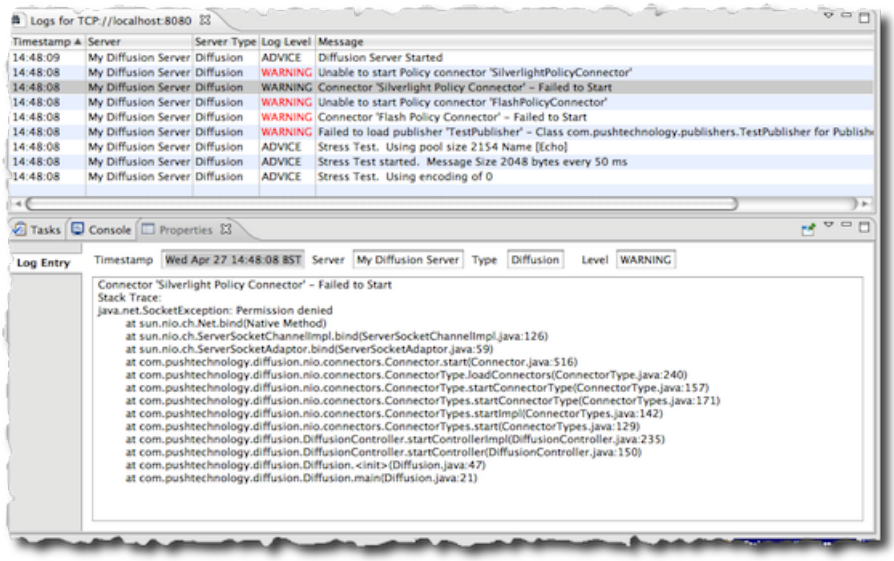

Logs for TCP://localhost:8080
Timestamp Server Server Type Log Level Message
14:48:09 My Diffusion Server Diffusion ADVICE Diffusion Server Started
14:48:08 My Diffusion Server Diffusion WARNING Unable to start Policy connector 'SilverlightPolicyConnector'
14:48:08 My Diffusion Server Diffusion WARNING Connector 'Silverlight Policy Connector' - Failed to Start
14:48:08 My Diffusion Server Diffusion WARNING Unable to start Policy connector 'FlashPolicyConnector'
14:48:08 My Diffusion Server Diffusion WARNING Connector 'Flash Policy Connector' - Failed to Start
14:48:08 My Diffusion Server Diffusion WARNING Failed to load publisher 'TestPublisher' - Class com.pushtechnology.publishers.TestPublisher for Publish
14:48:08 My Diffusion Server Diffusion ADVICE Stress Test. Using pool size 2154 Name [Echo]
14:48:08 My Diffusion Server Diffusion ADVICE Stress Test started. Message Size 2048 bytes every 50 ms
14:48:08 My Diffusion Server Diffusion ADVICE Stress Test. Using encoding of 0
Tasks Console Properties
Log Entry
Timestamp Wed Apr 27 14:48:08 BST Server My Diffusion Server Type Diffusion Level WARNING
Connector 'Silverlight Policy Connector' - Failed to Start
Stack Trace:
java.net.SocketException: Permission denied
at sun.nio.ch.Net.bind(Native Method)
at sun.nio.ch.ServerSocketChannelImpl.bind(ServerSocketChannelImpl.java:126)
at sun.nio.ch.ServerSocketAdaptor.bind(ServerSocketAdaptor.java:59)
at com.pushtechnology.diffusion.nio.connectors.Connector.start(Connector.java:516)
at com.pushtechnology.diffusion.nio.connectors.ConnectorType.loadConnectors(ConnectorType.java:240)
at com.pushtechnology.diffusion.nio.connectors.ConnectorType.startConnectorType(ConnectorType.java:157)
at com.pushtechnology.diffusion.nio.connectors.ConnectorTypes.startConnectorType(ConnectorTypes.java:171)
at com.pushtechnology.diffusion.nio.connectors.ConnectorTypes.startImpl(ConnectorTypes.java:142)
at com.pushtechnology.diffusion.nio.connectors.ConnectorTypes.start(ConnectorTypes.java:129)
at com.pushtechnology.diffusion.DiffusionController.startControllerImpl(DiffusionController.java:235)
at com.pushtechnology.diffusion.DiffusionController.startController(DiffusionController.java:150)
at com.pushtechnology.diffusion.Diffusion.<init>(Diffusion.java:47)
at com.pushtechnology.diffusion.Diffusion.main(Diffusion.java:21)

Figure 87: Server log entries

By clicking on the **Timestamp** table header, users can order the log entries by time-stamp. The pull down menu on the right hand side of the view header lets the user filter log entries by the entry log-level.

Note: This does not affect the logging from the server as this is a client-side filter only.

You can get the properties of server log entry from the context menu of an entry. Shown above are the properties of a multi-line log-entry.

Property obfuscator

This dialog is part of the Diffusion perspective and can be used to hide sensitive Diffusion configuration file entries, such as passwords and JMS login credentials.

Select or enter into the dialog the text that you want to obfuscate. All obfuscated entries start with `OB:`. The **Copy** button copies the obfuscated text to the clipboard, ready for pasting back into the configuration file.

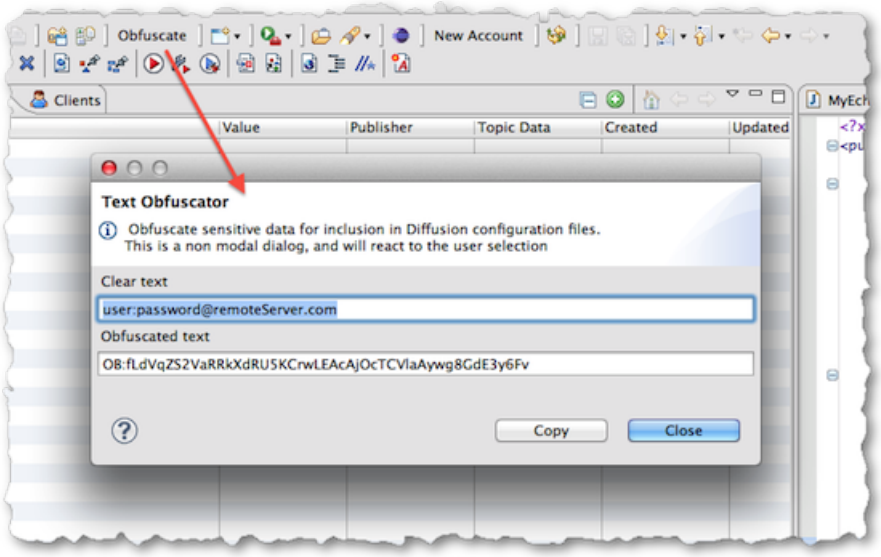


Figure 88: Property Obfuscator dialog

The dialog is modeless, and reacts to the users selection changes you do not need to dismiss the dialog if you have more than one value to obfuscate.

Chapter 21

Demos

In this section:

- Demos
- Building the demos using mvndar

Diffusion comes with demo applications that demonstrate certain features of Diffusion.

If you have chosen to include demos as part of the installation process, the demos are included as DAR files in the `demos` directory of your Diffusion installation and as source in the `demos/src` directory of your Diffusion installation. You can build the demos from source using *mvndar*.

Demos

The demos in the **Demos** section of the default Diffusion installation web page demonstrate various applications of Diffusion. They also contain a connection widget, allowing you to modify the transports used as well as inspect messages incoming and outgoing from the client.

To access the Diffusion web page and demos, start the web server and type in the following URL into a browser: http://localhost:8080.

Table 96: Demos provided with the Diffusion server

Tech Magnets demo	This demo provides an example of shared state handled by Diffusion. Tiles are created on the server in response to client actions. When a tile is moved by the client, a message is sent to the Diffusion server, the state is modified and an exclusive message is sent to all other subscribers. Open this demo in multiple browsers to see the real-time DOM updates.
Drawing Board demo	Multiple users draw on a virtual blackboard with colored chalk. All clients are updated in real-time with strokes from the other clients. With many strokes, there can be a large initial state which must be delivered to a new client, and the fragmented messages feature is used to handle this. Open this demo in multiple browsers to see the responsiveness of client interactions. Requires Canvas support in the browser.
Dogfight (game) demo	Aerial combat game for up to 8 players. Use the arrow keys to control your plane and the space key to fire. Uses HTML 5 features.
Asset Trader	View and trade on our cross-platform FX trading demo where price feeds, instruments, news, and user interactions are all distributed in real-time.

Building the demos using mvndar

You can use the Maven plugin *mvndar* with the provided `pom.xml` file to build the demos.

Building a DAR file using mvndar (recommended)

The mvndar plugin for Maven is developed and maintained on GitHub: https://github.com/pushtechnology/mvndar.

To use mvndar, reference it in the `plugins` section of your `pom.xml` file.

```
<plugins>
      ...
      <plugin>
        <groupId>com.pushtechnology.tools</groupId>
        <artifactId>dar-maven-plugin</artifactId>
        <version>1.2</version>
        <extensions>true</extensions>
      </plugin>
      ...
```

```
    </plugins>
```

The plugin runs during the project's package phase and creates the DAR file in the target directory.

Building the demos

The `pom.xml` file included in the `demos/src` directory of your Diffusion already contains the required reference to mvndar.

To build the demos using mvndar, complete the following steps:

1. From the command line, go to the *diffusion_installation*`/demos/src` directory.
2. Set the `DIFFUSION_HOME` environment variable to the absolute path of your Diffusion installation.
   - On Linux or OS X, type `export DIFFUSION_HOME=/`*diffusion_installation*
   - On Windows, type `set DIFFUSION_HOME=`*C*`:/`*diffusion_installation*
3. Run the `mvn clean install` command.

Related Links

Using Maven to build Java Diffusion applications on page 600
Apache Maven is a popular Java build tool and is well supported by Java IDEs. You can use Apache Maven to build your Diffusion applications.

Building publishers and other server application code with Maven on page 601
The Diffusion API for publishers and other server application code is not available in the Push Technology public Maven repository. To build server components, you must install the product locally and depend on `diffusion.jar` using a Maven system scope.

Chapter

22

Testing

In this section:

- Flex/Flash client
- Java client test tool
- Java event publisher test tool
- JavaScript client test tool
- Silverlight client test tool
- Windows client test tool (.NET)
- Windows event publisher test tool (.NET)
- Stress test tuning
- Benchmarking suite
- Test tools

This section covers some aspects of testing a Diffusion system.

Flex/Flash client

The Flex client test tool is a Flex application that uses the diffusion-flex.swc library. The tool can be found in the Tools section in the root menu of your Diffusion installation.

This tool enables the user to supply connection information as well as credential information. Select which transport is going to be used to connect to the Diffusion server. This client can issue a server ping. The ping response displays the elapsed time, its average and the current queue size for this client.

This tool can also be used to test reconnection, auto failover, and load balancing. Available servers can be listed by IP in the URL box. The **Reconnect** button, after a disconnect, returns the client to the server it was originally connected to, and receive messages queued during disconnection. Selecting **Auto Failover** causes the client to connect to the next available server, after the existing connection is forcibly killed. Selecting **Cascade** causes the client to cascade through the available servers until it finds one to which it can connect. Select **Load Balancing** in conjunction with **Auto Failover**, and the client connects to one of a shuffled list of available servers.

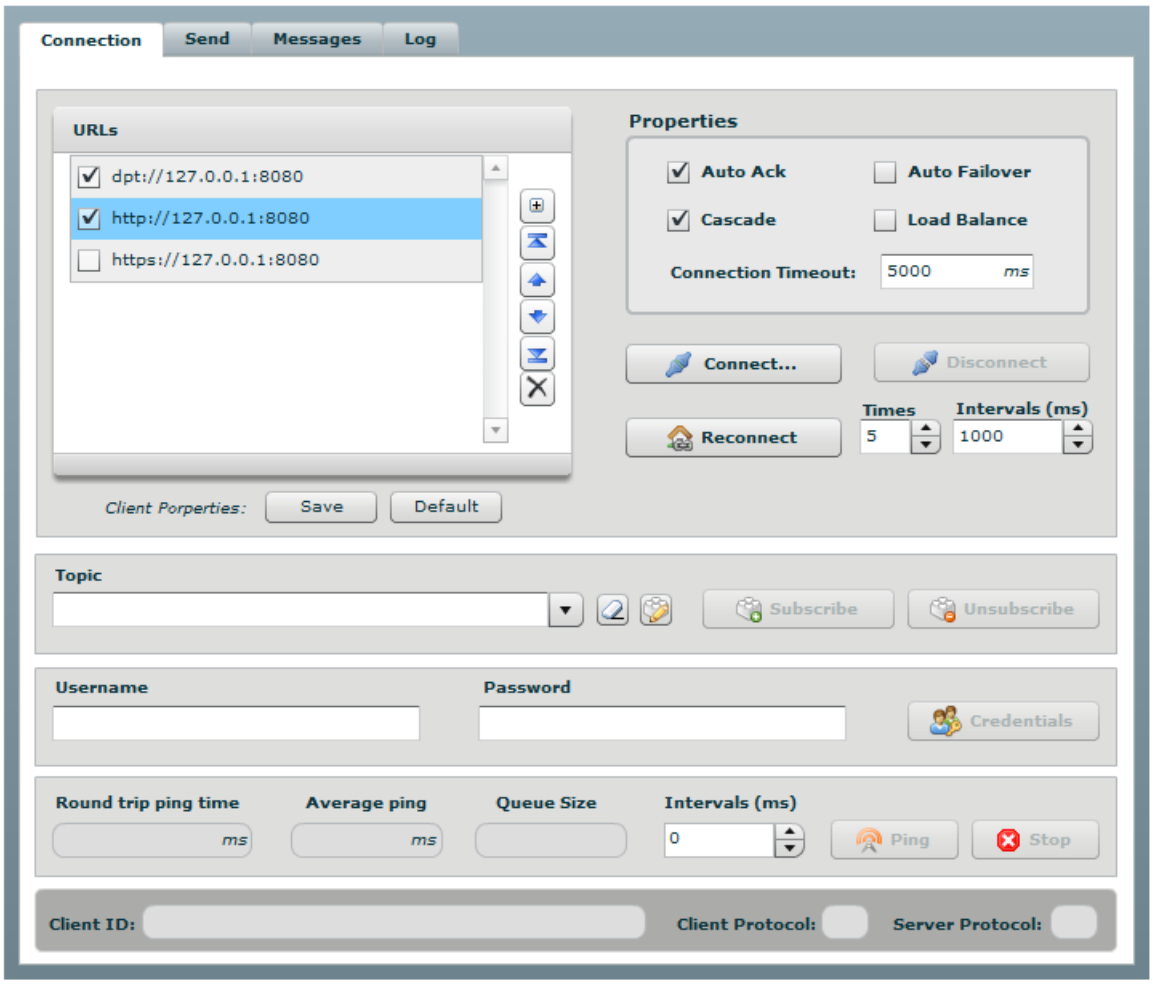


Figure 89: Flex client: Connection tab

You can pass user headers with a fetch request which are reflected back to the client in the response message as a Correlation ID. This allows the client to be able to associate replies with requests. The Flex client API has a new 'headers' parameter on the fetch methods. No changes are required to publishers to benefit from this.

Once connected to a Diffusion server, you can send messages to any topic. The publisher receives this message and notify the `messageFromClient` method in the publisher. The requested encoding can be set, as well as any user headers required. You can request an acknowledgment from the server.

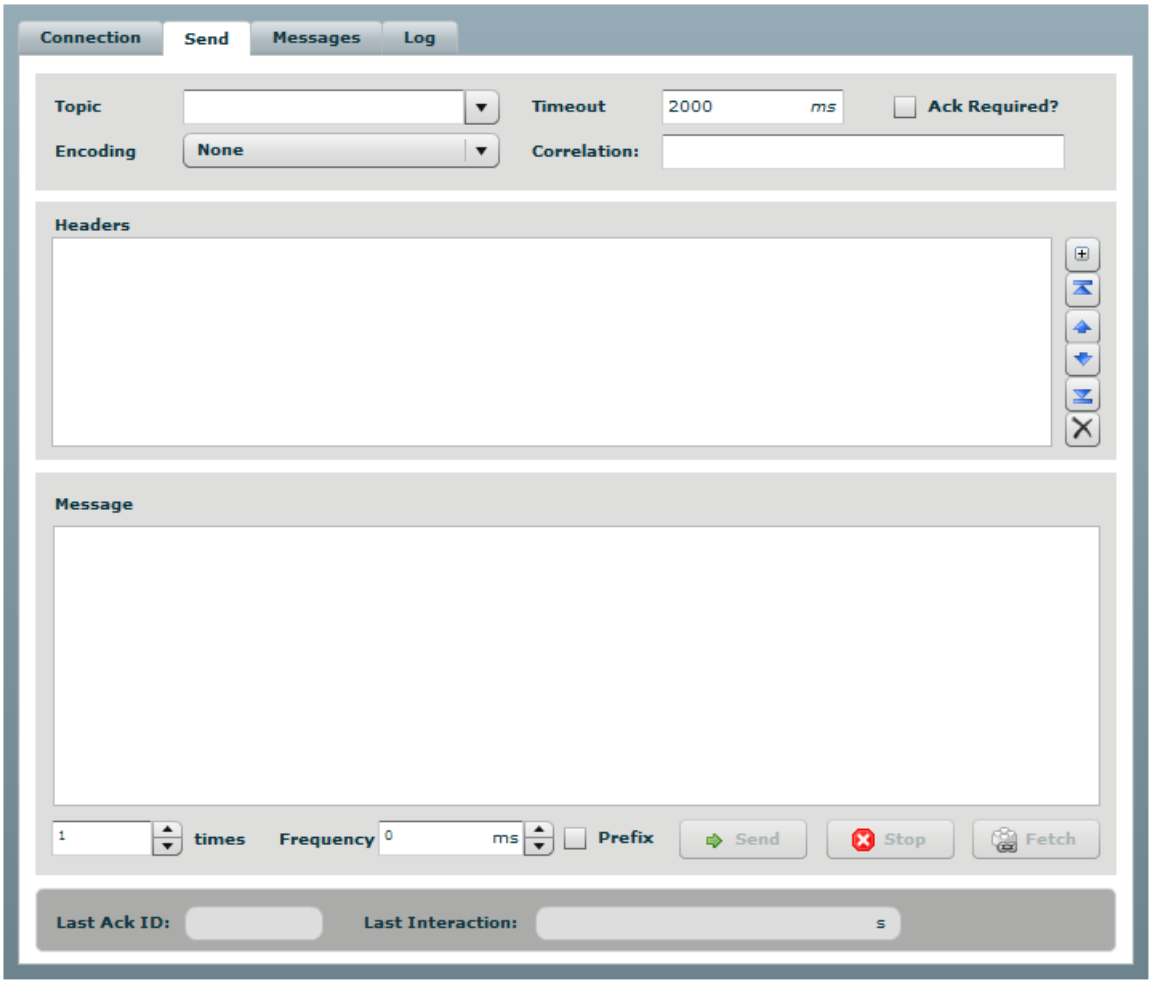


Figure 90: Flex client: Send tab

Messages are displayed in a tree format. It is not a true hierarchic tree, as each topic has its own node directly off the root node.

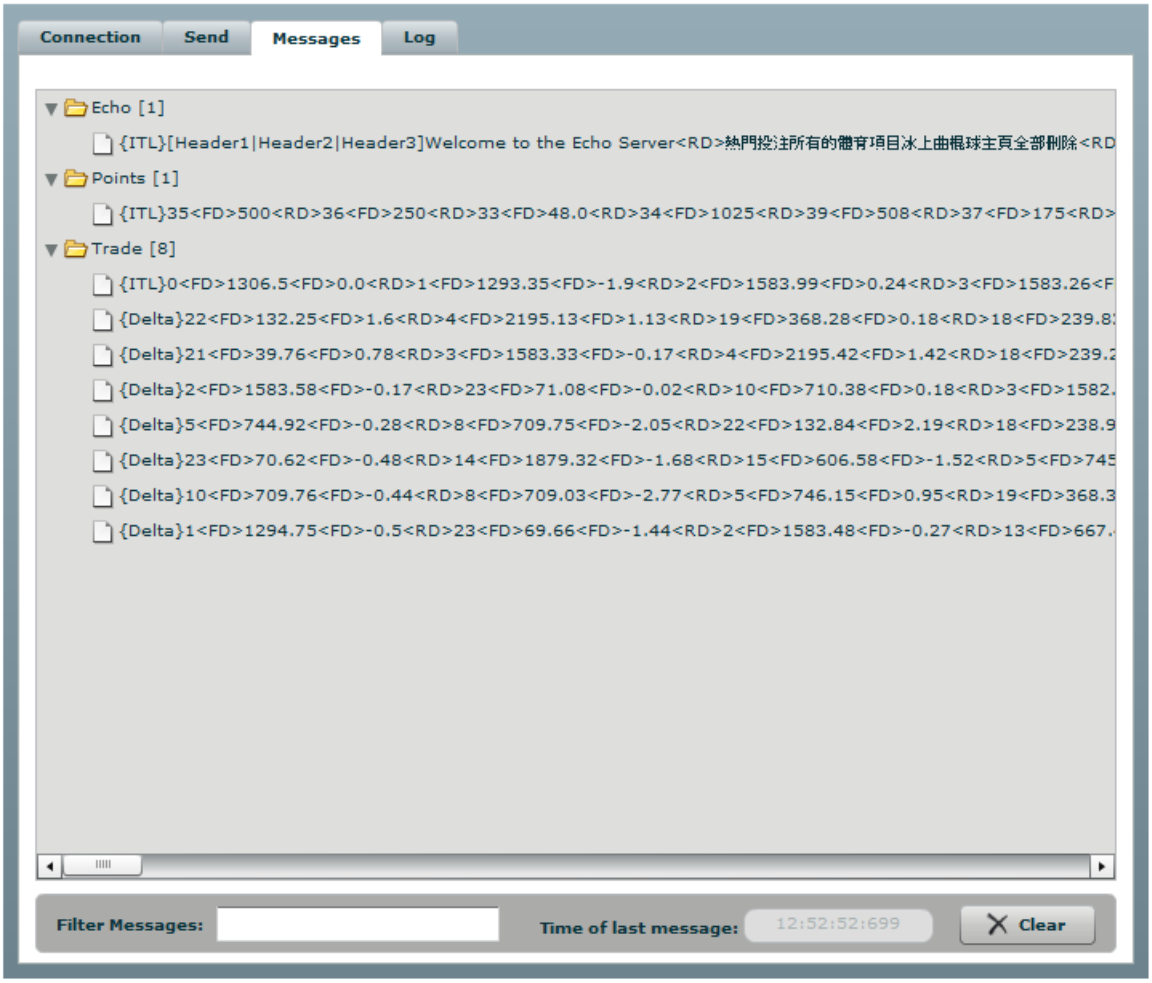
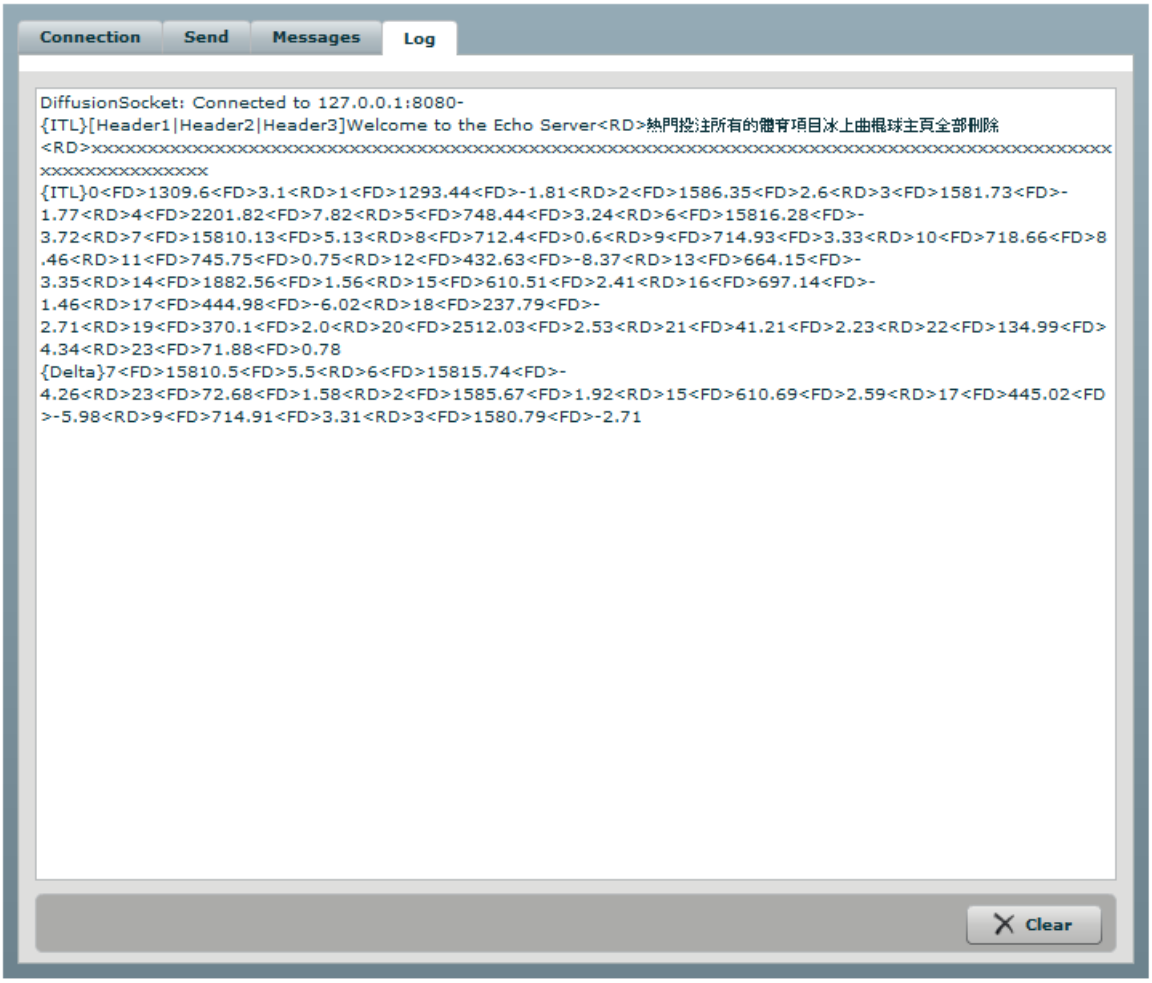


Figure 91: Flex client: Messages tab

The log tab shows messages that are sent from the Diffusion client when debugging is switched on.


Connection
Send
Messages
Log
DiffusionSocket: Connected to 127.0.0.1:8080-
{ITL}[Header1|Header2|Header3]Welcome to the Echo Server<RD>熱門投注所有的體育項目冰上曲棍球主頁全部刪除
<RD>xxxx

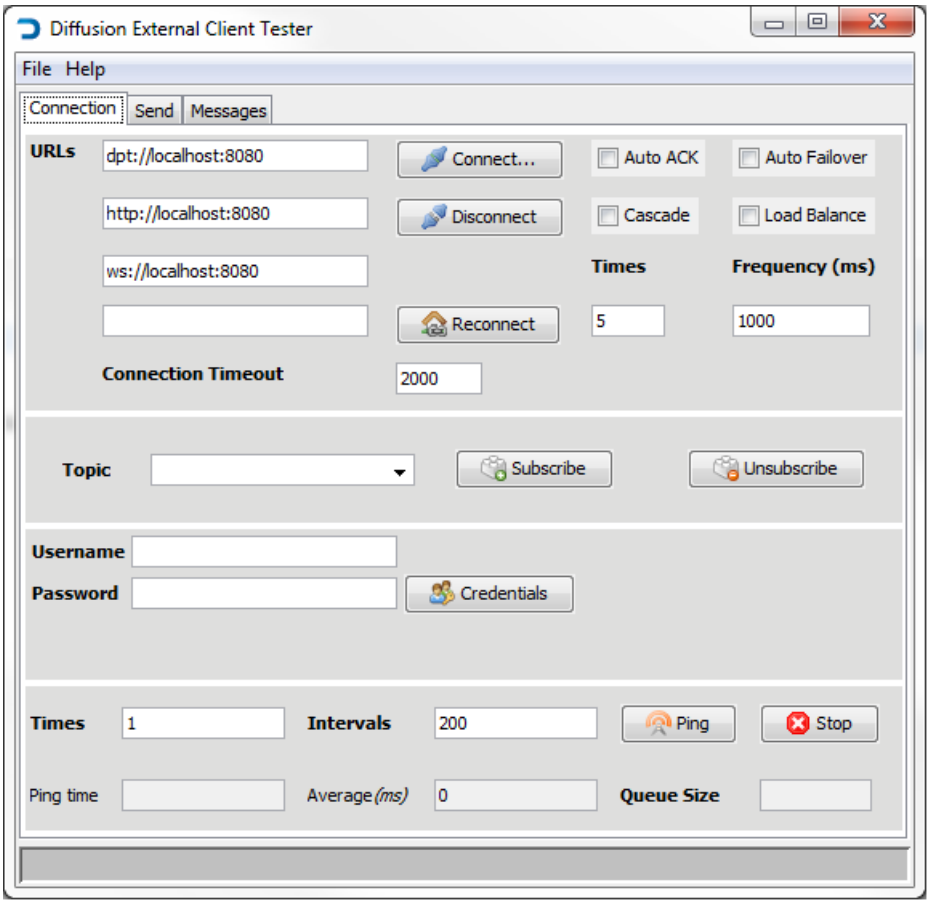


Figure 93: External client tester: Connection tab

You can connect to Diffusion securely. In order for the external client to work in secure mode, a certificate must be installed on the client machine. To aid in installing the certificate, there is an option under the file menu called Load Cert. This connects to Diffusion and detail instructions on how to install the newly created jssecerts file.

After subscribing to a topic, the client can send messages to that topic. The publisher receives this message and notify the `messageFromClient` method in the publisher. It is also possible to set the message encoding type and any user headers if required.

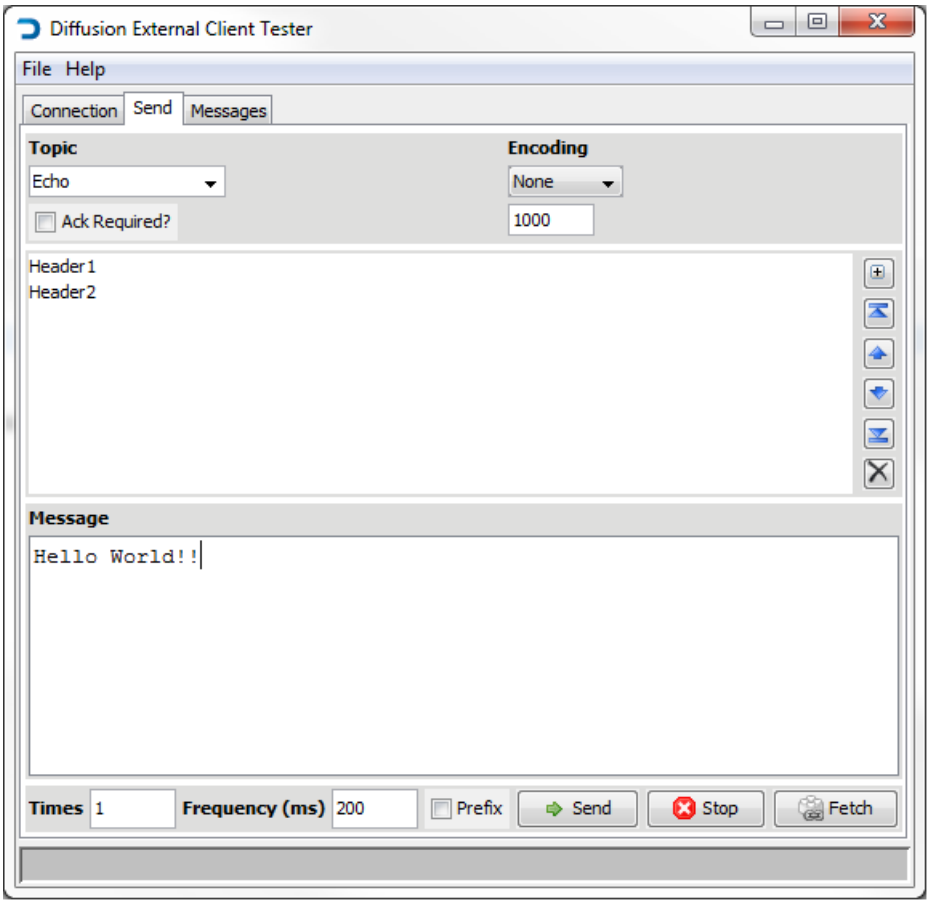


Figure 94: External client tester: Send tab

Messages are displayed in a tree format. It is not a true hierarchic tree, as each topic has its own node directly off the root node.

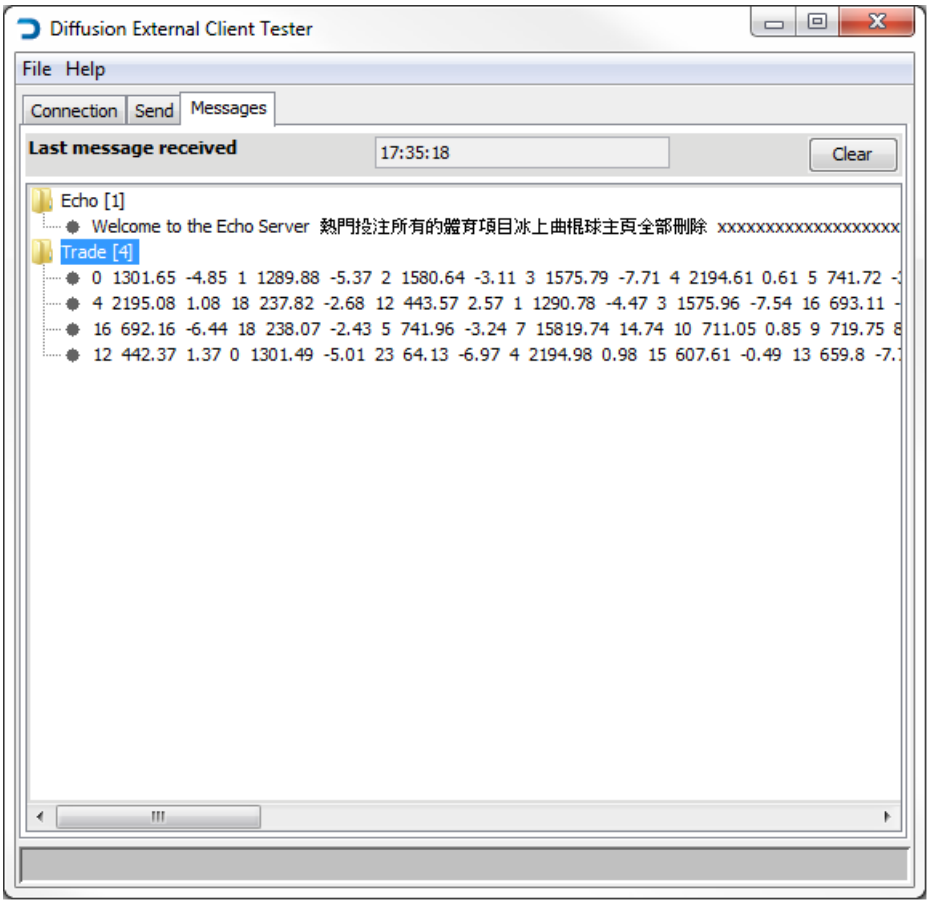


Figure 95: External client tester: Messages tab

It is also possible to examine a message by double clicking it

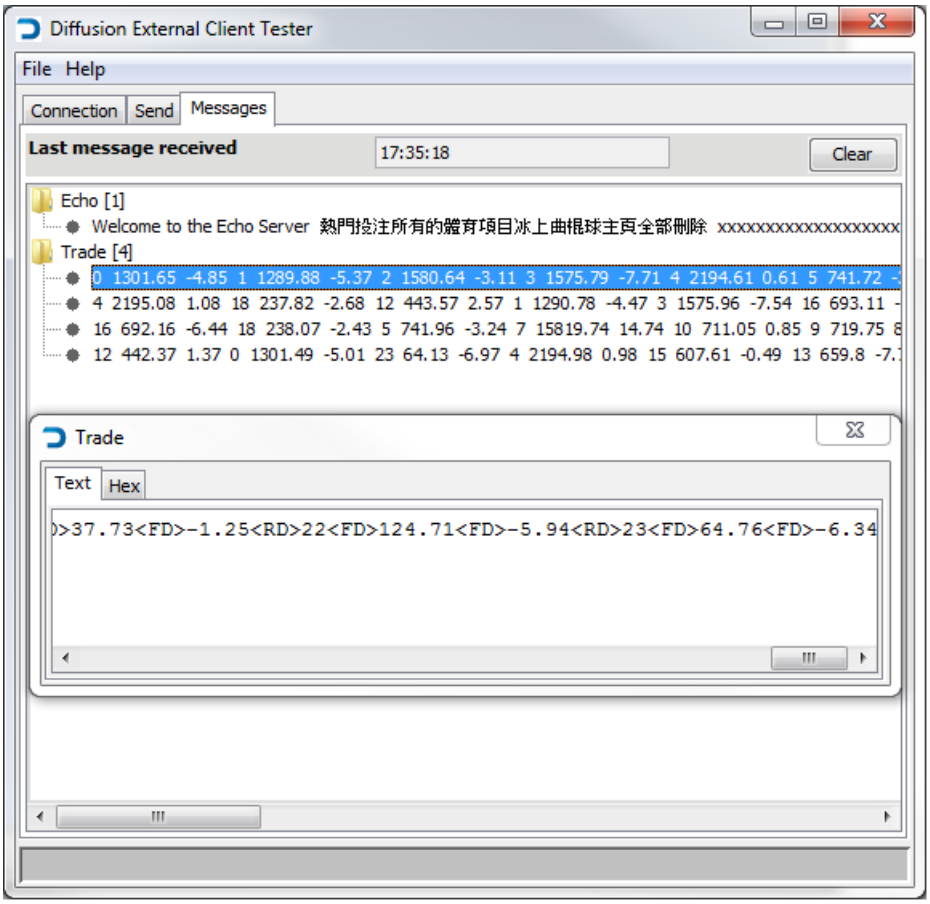


Figure 96: External client tester: Message details window

Java event publisher test tool

The event publisher test tool is located in the `Tools` folder of your Diffusion installation, and is run from the command line using either `eventpub.sh` or `eventpub.bat`.

It enables messages to be sent to a Diffusion server. This basically enables the user to send messages to the publisher that is managing the topic specified.

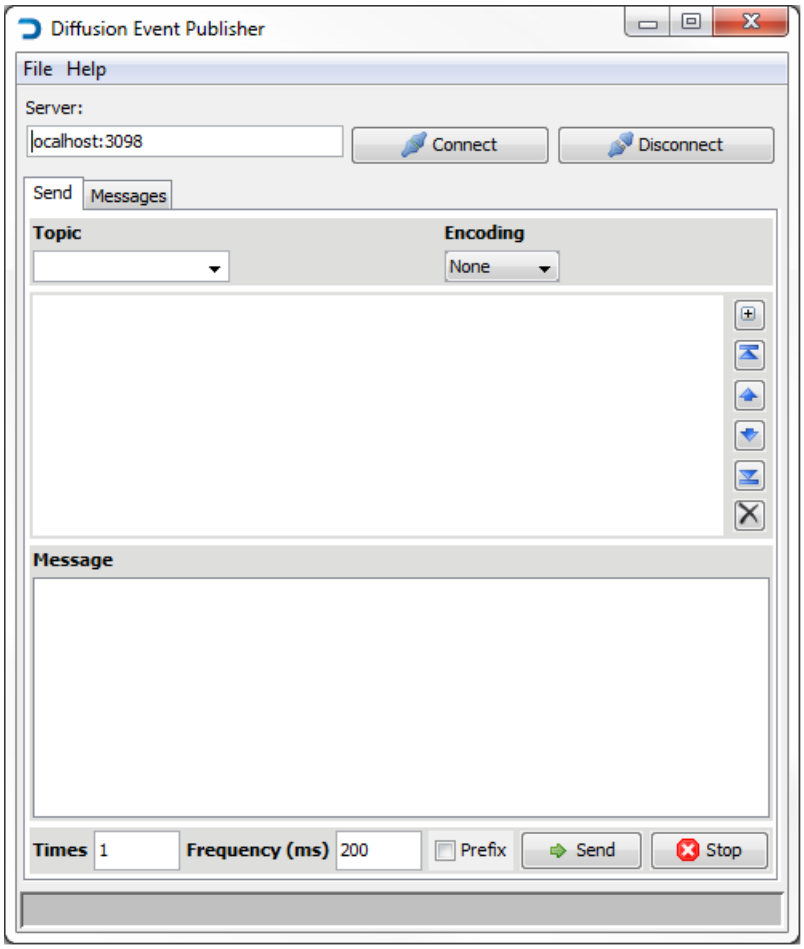


Figure 97: Event publisher test tool: Send tab

The publisher gets notified of the message on the `messageFromEventPublisher` method.

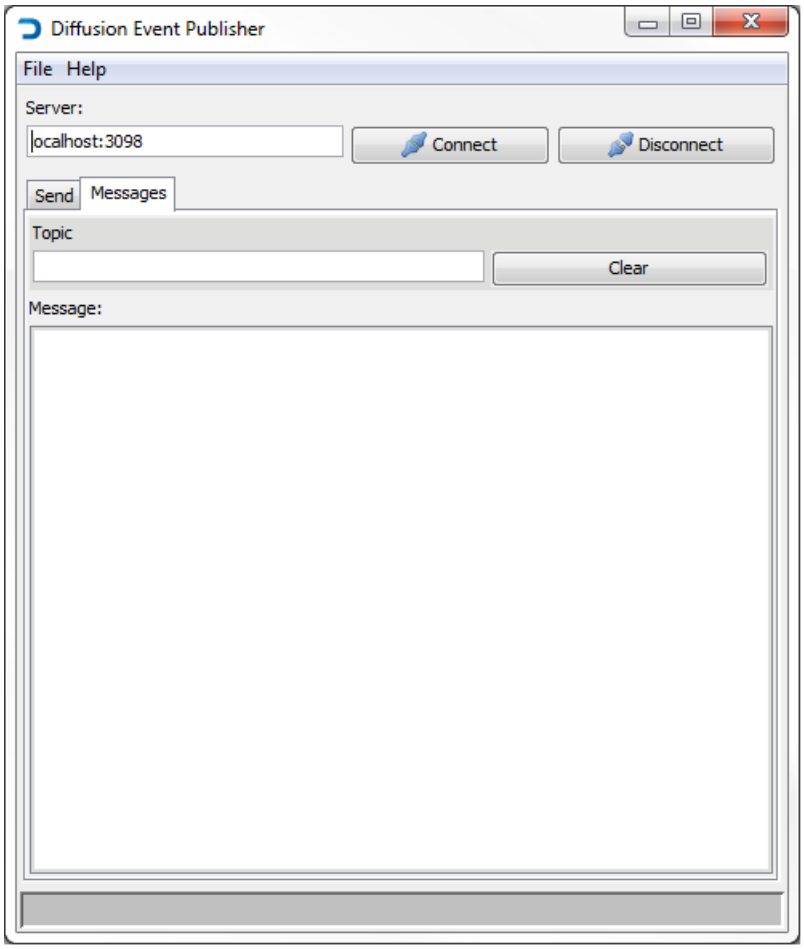


Figure 98: Event publisher test tool: Messages tab

JavaScript client test tool

The JavaScript test tool allows the user to connect to a Diffusion publisher and test the API.

The JavaScript test tool is browser based and can be found in the Root Menu of your Diffusion installation. It allows for different types of connections. The test tool also allows for Subscribing, Unsubscribing and Fetching against topic sets.

Fetch Correlation allows user headers to be passed with a fetch request which is reflected back to the client in the response message. This allows the client to be able to associate replies with requests. The JavaScript client API has a new headers parameter on the fetch methods. See API documentation for changes to the API specification. No changes are required to publishers to benefit from this.

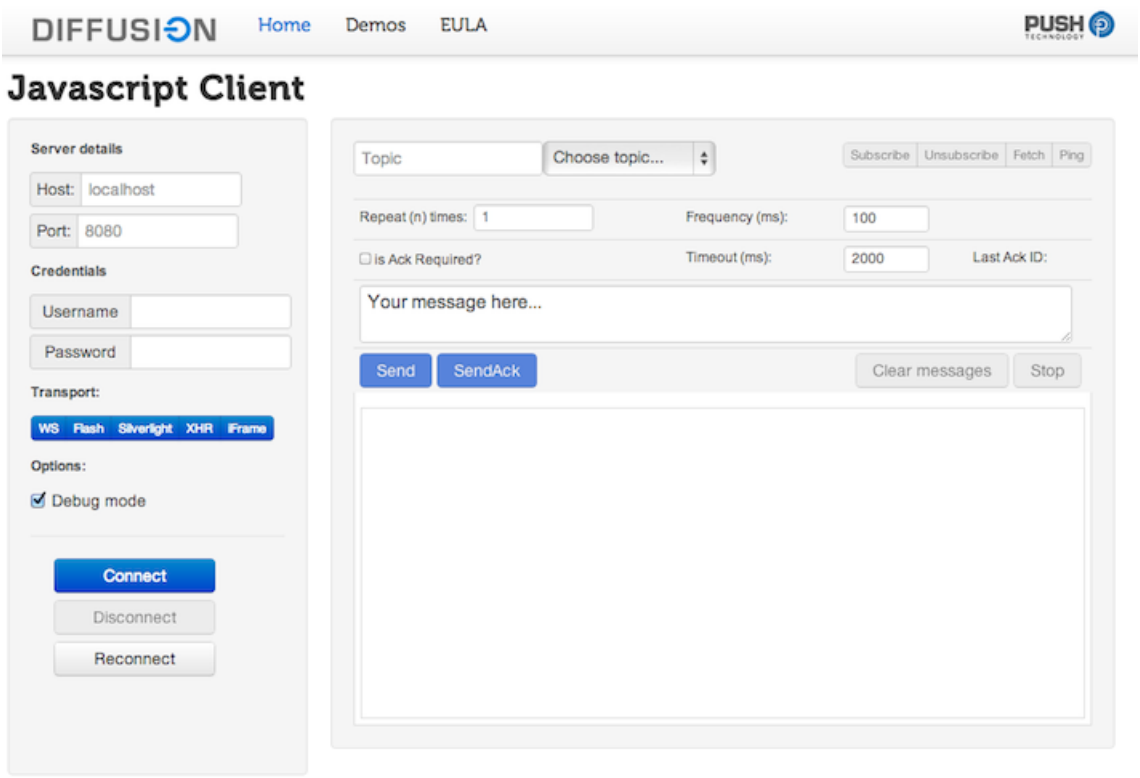


Figure 99: JavaScript test tool

Silverlight client test tool

The Silverlight client test tool is a Silverlight application that uses the `PushTechnology.Transports` assembly. This tool enables the user to perform various diagnostic functions without having to build a new Silverlight application.

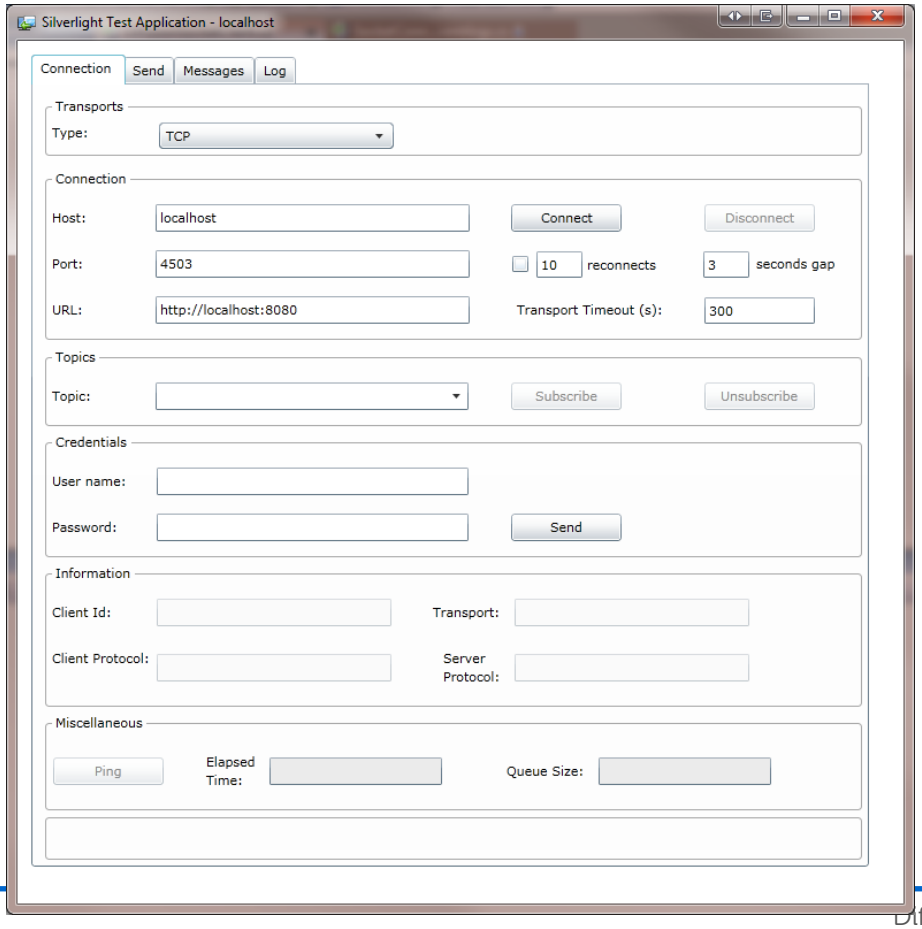


Figure 100: Silverlight test tool: Connection tab

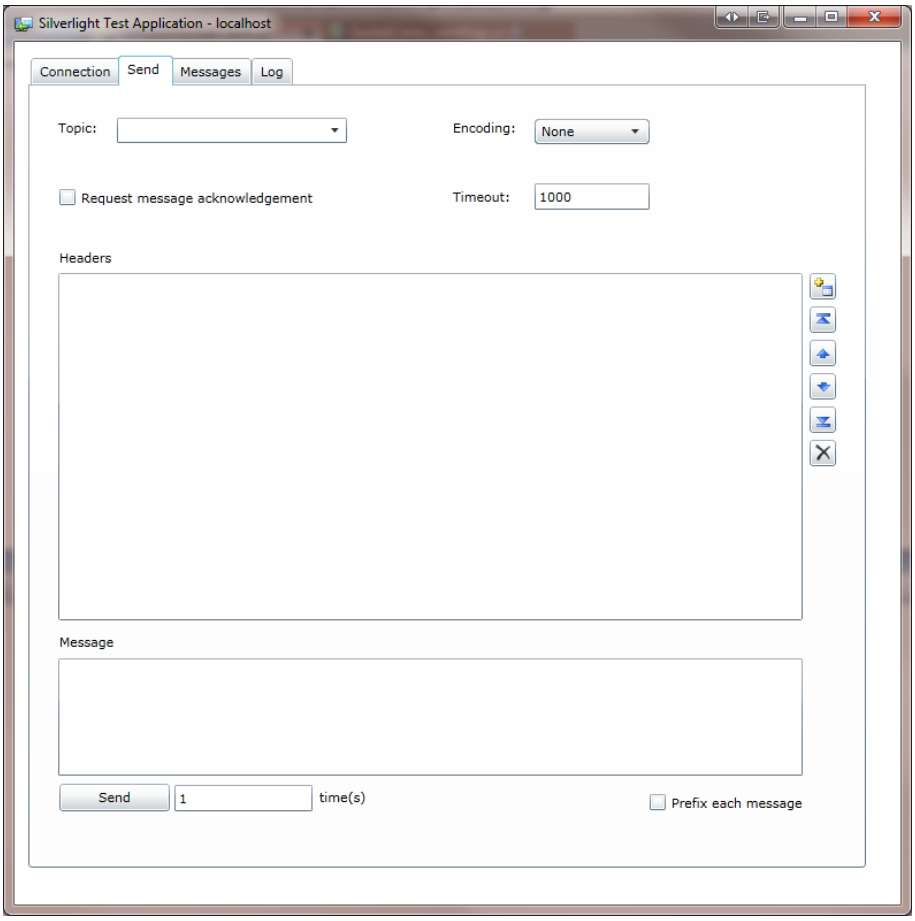


Figure 101: Silverlight test tool: Send tab

Received messages are displayed in a hierarchical format:

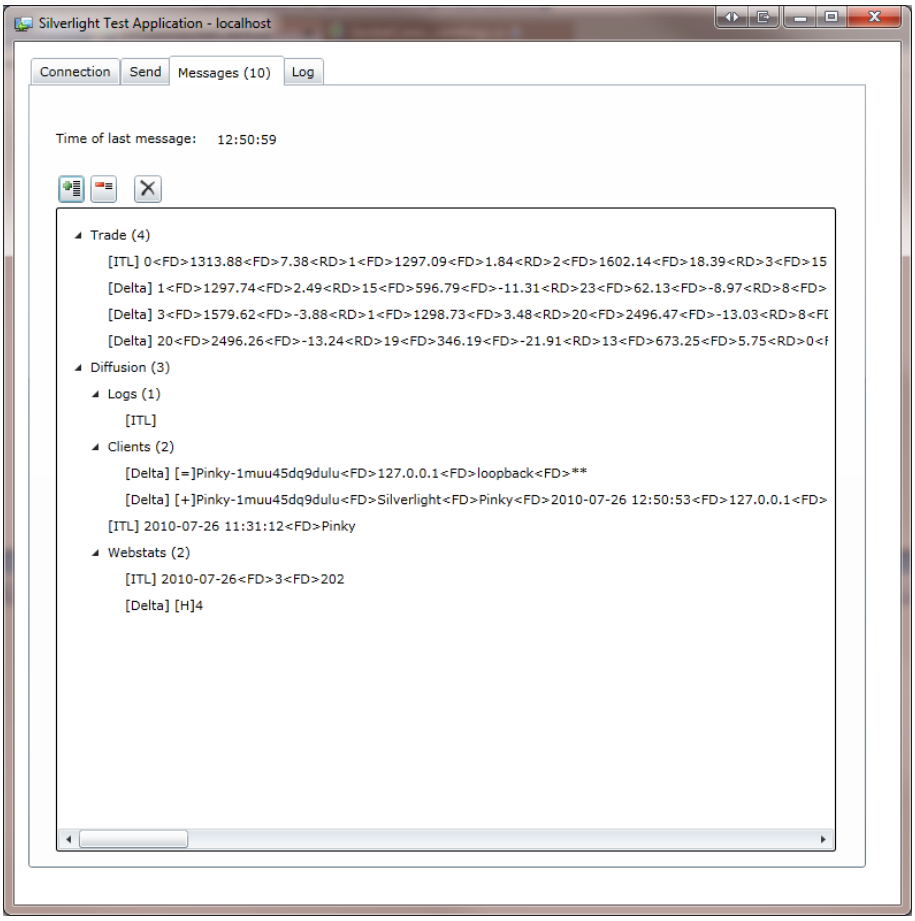


Figure 102: Silverlight test tool: Messages tab

Windows client test tool (.NET)

The external client test tool is a .NET application that enables the user to connect/subscribe/unsubscribe to a Diffusion server. It allows for many different connection types (for example, HTTP/DPTS), and also allows for the composition and sending of messages.

Below is the **Connection** tab of the external client test tool; this contains all of the facilities required to establish a connection to the Diffusion server.

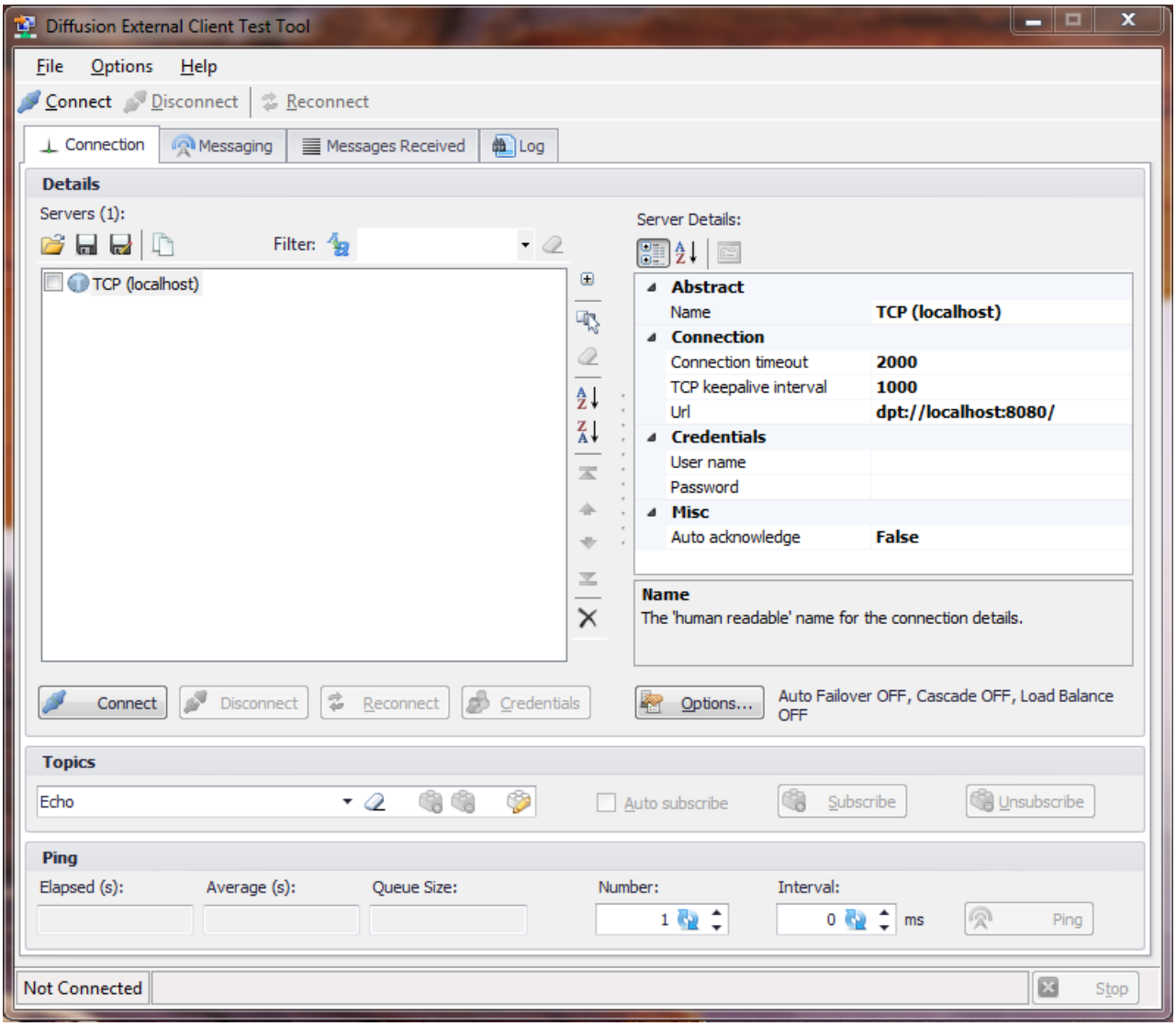


Figure 103: Windows client test tool: Connection tab

Additional topics can be subscribed at any time subsequent to connection by entering the topic names in the topics field and clicking the **Subscribe** button. The converse action (unsubscription) can be performed with the **Unsubscribe** button.

You can issue a server ping from this client. The elapsed time is indicated in the relevant fields, along with a rolling average.

If connection credentials are required to subscribe to a particular topic, these can be entered in the **Credentials** section. They can also be sent retrospectively using the **Credentials** button.

Various connection options are available to the user by clicking the **Options** button:

Auto failover

After a connection to Diffusion is placed, if the connection fails and if there is a pool of server details to pick from, the next set of server details is chosen and connection placed to the next server.

Cascade

When the client software attempts to place a connection, if the connection attempt fails, the next set of server details in the list is chosen and used. It is similar to Auto Failover except this logic is applied prior to a connection, whereas Auto Failover is applied once a connection is in place.

Load balance

If **Load Balance** is set, the list of server details is shuffled prior to use.

Restore state when reconnecting

If connection to Diffusion is lost, and the server has been configured accordingly, the client reconnects and receives any messages that have been missed when the connection failed.

Once subscribed to a topic, you can send messages to either that specific topic, or any registered topic, using the **Send** tab:

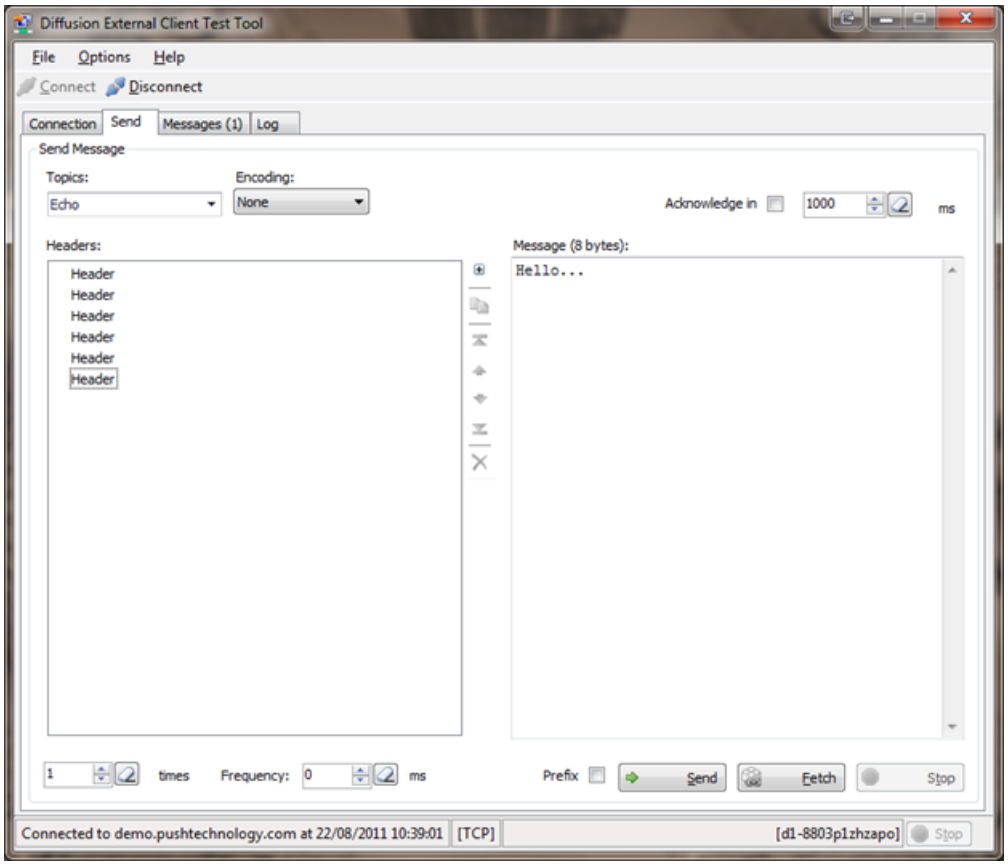


Figure 104: Windows client test tool: Send tab

As can also be seen, you can compose a message containing headers; these can be added using the convenient toolbar to the side of the main headers area. You can also send a message multiple times by incrementing the edit field at the bottom of the message composition area.

Any received messages from the Diffusion server are shown in the **Messages** tab as shown in the following figure:

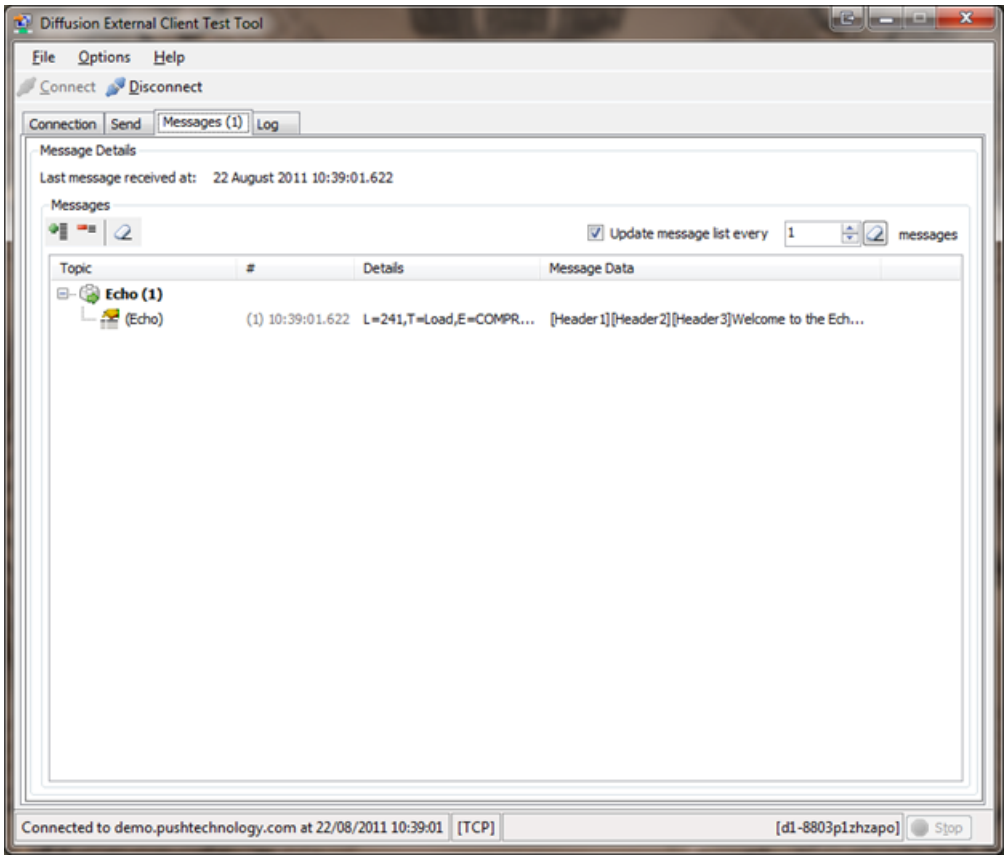


Figure 105: Windows client test tool: Messages tab

Messages are displayed in a true hierarchical format, so deeply-nested topics will be relatively easy to locate and analyze.

If a high volume of messages is expected from the Diffusion server, you can filter the number of messages by adjusting the **Update message list** value. You can disable message updates completely by clicking the check box.

You can view logging information generated by the .NET libraries:

The logging type and level can be adjusted using the **Preferences** option on the main menu.

Windows event publisher test tool (.NET)

The event publisher test tool enables messages to be sent to a Diffusion server. This basically enables the user to send messages to the publisher that is managing the topic specified.

The event publisher test tool enables messages to be sent to a Diffusion server. This enables the user to send messages to the publisher that is managing the topic specified.

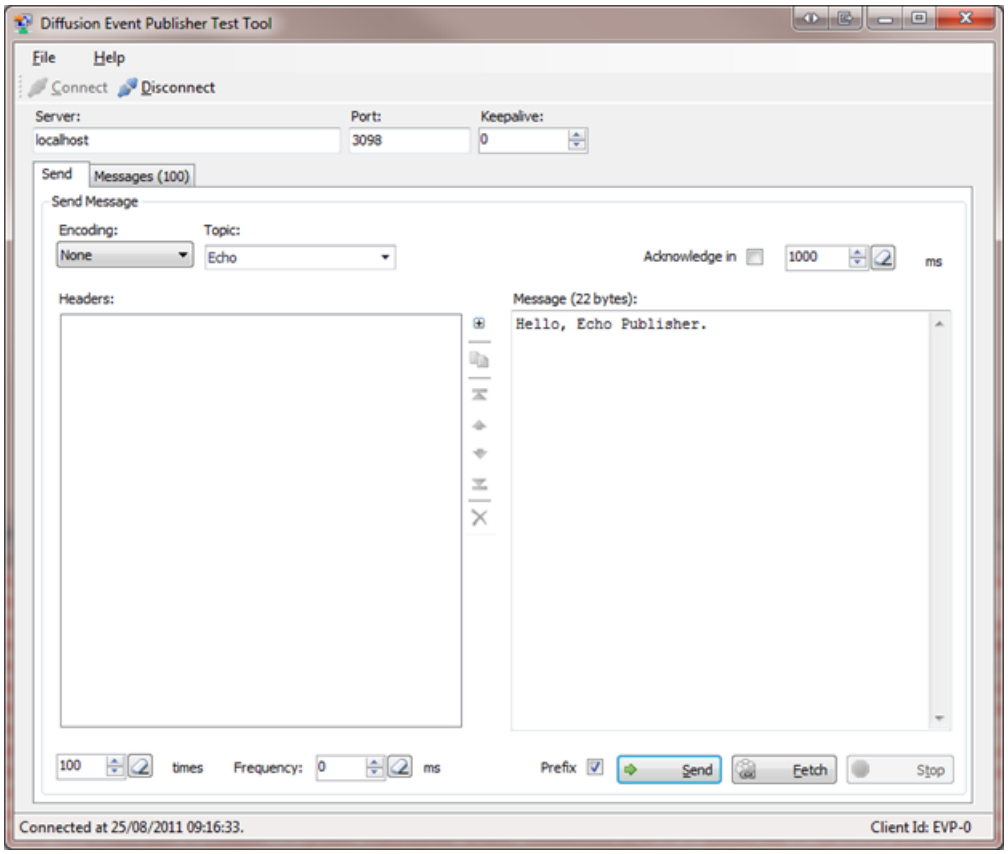


Figure 106: Windows event publisher test tool: Send tab

As an event publisher is bi-directional, any messages received are displayed in the **Messages** tab:

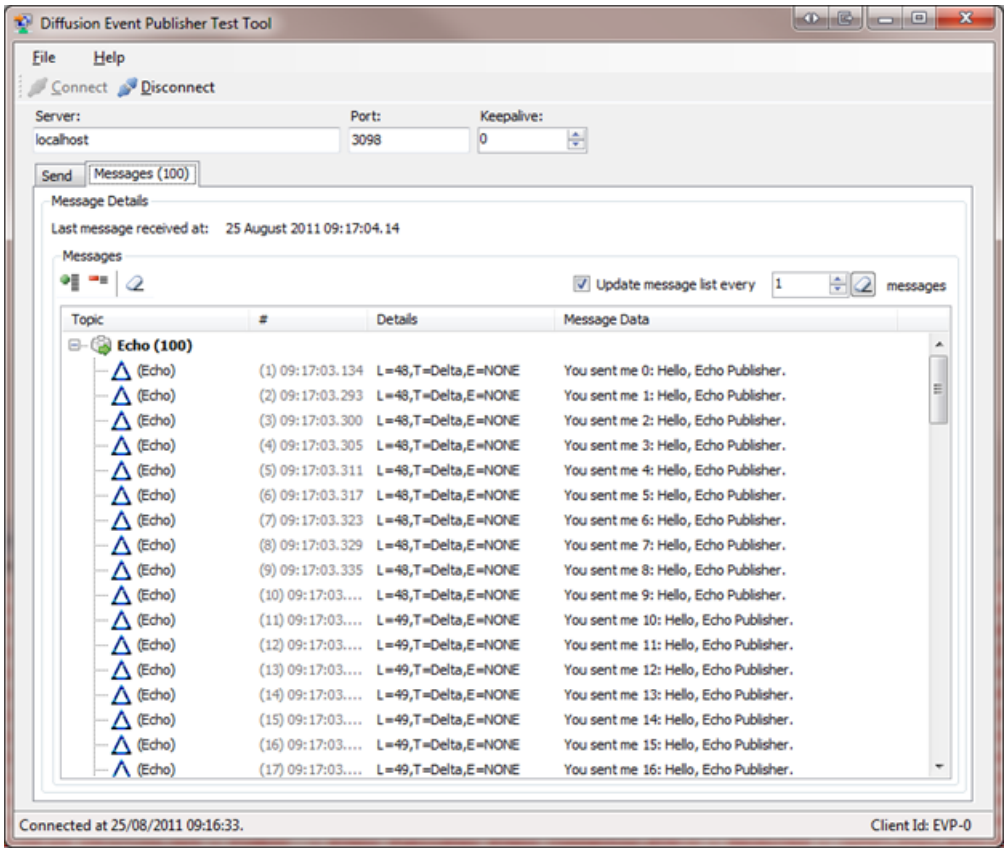

Diffusion Event Publisher Test Tool
File
Help
Connect
Disconnect
Server:
localhost
Port:
3098
Keepalive:
0
Send
Messages (100)
Message Details
Last message received at: 25 August 2011 09:17:04.14
Messages
Update message list every
1
messages
Topic
#
Details
Message Data
Echo (100)
(Echo) (1) 09:17:03.134 L=48,T=Delta,E=NONE You sent me 0: Hello, Echo Publisher.
(Echo) (2) 09:17:03.293 L=48,T=Delta,E=NONE You sent me 1: Hello, Echo Publisher.
(Echo) (3) 09:17:03.300 L=48,T=Delta,E=NONE You sent me 2: Hello, Echo Publisher.
(Echo) (4) 09:17:03.305 L=48,T=Delta,E=NONE You sent me 3: Hello, Echo Publisher.
(Echo) (5) 09:17:03.311 L=48,T=Delta,E=NONE You sent me 4: Hello, Echo Publisher.
(Echo) (6) 09:17:03.317 L=48,T=Delta,E=NONE You sent me 5: Hello, Echo Publisher.
(Echo) (7) 09:17:03.323 L=48,T=Delta,E=NONE You sent me 6: Hello, Echo Publisher.
(Echo) (8) 09:17:03.329 L=48,T=Delta,E=NONE You sent me 7: Hello, Echo Publisher.
(Echo) (9) 09:17:03.335 L=48,T=Delta,E=NONE You sent me 8: Hello, Echo Publisher.
(Echo) (10) 09:17:03.... L=48,T=Delta,E=NONE You sent me 9: Hello, Echo Publisher.
(Echo) (11) 09:17:03.... L=49,T=Delta,E=NONE You sent me 10: Hello, Echo Publisher.
(Echo) (12) 09:17:03.... L=49,T=Delta,E=NONE You sent me 11: Hello, Echo Publisher.
(Echo) (13) 09:17:03.... L=49,T=Delta,E=NONE You sent me 12: Hello, Echo Publisher.
(Echo) (14) 09:17:03.... L=49,T=Delta,E=NONE You sent me 13: Hello, Echo Publisher.
(Echo) (15) 09:17:03.... L=49,T=Delta,E=NONE You sent me 14: Hello, Echo Publisher.
(Echo) (16) 09:17:03.... L=49,T=Delta,E=NONE You sent me 15: Hello, Echo Publisher.
(Echo) (17) 09:17:03.... L=49,T=Delta,E=NONE You sent me 16: Hello, Echo Publisher.
Connected at 25/08/2011 09:16:33.
Client Id: EVP-0

Figure 107: Windows event publisher test tool: Messages tab

Stress test tuning

Stress testing Diffusion requires various tuning changes to be made.

Table 97: Tuning changes for stress testing

Increase number of open files	If the Diffusion server is running in Linux, there is a limit to the number of open files/ connections. See "Increasing Number of Open files".
Increase number of acceptors	Performance gains can be obtained by increasing the number of acceptors. (The number of physical acceptors that handles incoming requests). For Flash clients, the value in `etc/Connectors.xml` must be increased. Increase these values carefully and the server re-tested each time. Depending on the tests, there is a point where the increase has a detrimental effect. It is advisable to make small increases then re-test (for example, 5-10).
Buffer sizes	Socket buffer sizes can be changed in `etc/Connectors.xml`
Inbound thread settings	The Inbound thread pool settings must be tuned otherwise the pool and queues get full.

	The settings are found in `etc/Server.xml`. Increase these values carefully and the server re-tested each time. Set the maximum size and queue size quite high for stress testing (for example, 250) so that the server can handle the hundreds or thousands of simultaneous connection attempts.
Client queues	Depending on the number and frequency of the messages, you might have to increase the client queue parameter.
Client multiplexers	By default, there are two client multiplexers configured. You might have to increase the number of multiplexers as the client load on each Diffusion server increases.

Custom stress test

The supplied stress test package looks at the `etc/Publishers.xml` and stresstest.properties for its configuration. As the source of the stress test package is supplied there is nothing to stop the user extending the supplied stress test package as a framework for their own custom tests. It is worth stress testing the publisher that you will use in production as this is going to show any performance degradation under duress, and also highlight any data synchronicity issues. It might also be worth changing the default behavior of the clients as well the client connection ratio that you expect. For instance there is no point in testing HTTP clients, if your solution is going to be Flash only, but if your solution is going to be JavaScript, then it is worth setting up a ratio of plug-in (Flash) clients and HTTP clients. If your solution is going to be solely SSL, it is only worth configuring the SSL clients.

Limitations

If more than 2000 clients are to be simulated, it is better to run the stress test from multiple machines. Most systems start to struggle when dealing with 2000 concurrent connections. By running the stress test on multiple machines any number of clients can be tested.

Socket exceptions can be encountered when a large number of clients are run on one machine. These can be thrown on either the Diffusion server or within the stress test tool. If this is occurring, the number of clients needs to be reduced

Stress test

Stress test is a good way of working out what the system requirements are going to be of a Diffusion server under duress. The stress test tool is issued to test the performance of Diffusion under stress and helps to understand how changes to the configuration can have an effect upon the performance.

Before you begin

Do not run the stress test tool on the same machine as the Diffusion server. A separate machine is required to run the stress test tool.

About this task

Installing the stress test tool;

Procedure

1. On the test machine, create a `stresstest` directory and copy the files `stresstest/stresstest.bat`, `stresstest/stresstest.sh`, `stresstest/stresstest.properties` and `lib/diffusion.jar` from the Diffusion installation

2. Configure stress test by editing the `stresstest.properties` file. The following need to be configured:
3. Number of clients
4. Transport type
5. Server details
6. Number of messages with which to run the test
7. The topic to which the clients subscribe
8. On a Windows system, launch `stresstest.bat`, on a UNIX machine launch `stresstest.sh`.

Benchmarking suite

A benchmarking suite for Diffusion is available on GitHub. You can use this suite to test the latency and throughput of publishers.

The benchmarking suite is available at the following location: https://github.com/pushtechnology/diffusion-benchmark-suite.

The benchmarking suite works on Linux only and requires the following software be installed on the system:

- Apache Ant
- Java with JDK
- Diffusion server

For more information about using the benchmarking suite, see the readme file in the GitHub project.

Test tools

Test tools are provided which allow you to connect to a Diffusion server as an external client or event publisher application. There are also tools that act as a publisher server application.

Generic Java versions are provided which work on any platform and Windows versions are provided specifically for use on Windows platforms.

Table 98: Testing tools

Java client	Provides a GUI interface simulating an external client connection using the Java API. Suitable for use on any platform.
Java event publisher	Provides a GUI interface simulating an event publisher connection using the Java API. Suitable for use on any platform.
Windows client	Provides a GUI interface simulating an external client connection using the Windows API. Suitable for use on Windows platforms only.
Windows event publisher	Provides a GUI interface simulating an event publisher connection using the Windows API. Suitable for use on Windows platforms only.
JavaScript client	Provides a GUI interface simulating a JavaScript client connection. Suitable for use on any platform.

Flex/Flash client	Provides a GUI interface simulating a Flash client connection. Suitable for use on any platform where Flash Player is installed.
Silverlight client	Provides a GUI interface simulating a Silverlight client connection. Suitable for use on any platform where Silverlight is installed.

Chapter 23

Tools

In this section:

- Tools for Amazon Elastic Compute Cloud (EC2)
- Tools for Joyent
- Using Maven to build Java Diffusion applications

If the tools were installed during the installation process, there are some tools that can help with the monitoring of Diffusion plus some handy utilities.

There are additional tools and utilities that are available in public repositories, such as GitHub and Maven.

Tools for Amazon Elastic Compute Cloud (EC2)

This is a description of a number of tools and adaptations which are provided for use when using Diffusion with Amazon EC2™.

Diffusion includes in `tools/ec2/` a number of files useful when deploying Diffusion to an Ubuntu image on an EC2 virtual machine.

diffusion.conf

Ubuntu makes use of the Upstart daemon as a replacement for `init`. Copied to `/etc/init/diffusion.conf` this file contains configuration for Upstart to begin Diffusion at boot-time in a background process as a unprivileged user. It establishes iptables rules to route traffic from privileged ports to Diffusion.

Users can stop and start Diffusion using Upstart commands, for example, to start the server

```
service diffusion start
```

To stop Diffusion

```
service diffusion stop
```

To check the status of Diffusion

```
service diffusion status
```

In the event that something goes amiss Upstart writes a log file to `/var/log/upstart/diffusion.log`

etc/Connectors.xml

Except for two port number changes this is an otherwise regular copy of `etc/Connectors.xml` normally found in a Diffusion installation. This edition however binds the Flash policy connector and the Silverlight policy connector to port numbers greater that 1024, making it possible to run Diffusion as an unprivileged process. Use this file in conjunction with the iptables rules established in `diffusion.conf`

ec.xml

An illustrative Apache Ant script that can be used to start, stop, and get status from a Diffusion server running on an Amazon EC2 Linux host. It also demonstrates an inelegant means of deploying and undeploying a DAR file to/from the remote server, by copying the file to the remote server, then moving it into the deploy directory.

Table 99: Targets

Property	Purpose
start	Runs `sudo service diffusion start` on remote host using SSH
stop	Runs `sudo service diffusion stop` on remote host using SSH
status	Runs `sudo service diffusion status` on remote host using SSH
deploy	Uploads local DAR file to staging dir, then moves it into Diffusion deploy directory

Property	Purpose
undeploy	Removes DAR file from deploy directory, signaling Diffusion to undeploy related publishers (where possible)

The script is driven by named properties:

Table 100: Properties for targets start, stop and status

Property	Purpose
remote.host	EC2 host running sshd
remote.username	Authentication username, default of `ubuntu`
remote.keyfile	PEM encoded key use during authentication

Table 101: Additional properties for targets deploy and undeploy

Property	Purpose
dar.file	Name of a DAR file to deploy or undeploy
remote.diffusion.dir	Root directory of the remote Diffusion installation

Example deployment:

```
ant -Dremote.host=54.235.65.36 \
 -Dremote.keyfile=$HOME/.ssh/ec2-push1.pem \
 -Ddar.file=$HOME/Applications/Diffusion5.1.0/demos/dogfight.dar \
 -Dremote.diffusion.dir=/home/ubuntu/Diffusion/Diffusion5.1.0 \
 deploy
```

The script uses no proprietary Diffusion code and is open to extension during development of a solution. Both the `sshexec` and `scp` tasks depend on the jsch library which might have to be downloaded.

Tools for Joyent

This is a description of adaptations which are provided for use when using Diffusion within the Joyent Cloud.

Diffusion includes in `tools/joyent/` files useful when deploying Diffusion to an SmartOS virtual machine in the Joyent cloud.

`diffusion.xml`

SmartOS makes use of SMF to start services at boot-time. Once tailored for the host and installed into SMF this file contains configuration for SMF to begin Diffusion at boot-time in a background process as an unprivileged user called `push`. This file does not create or assert the existence of a `push` user.

Users can stop and start Diffusion using SMF commands. For example, to start the server

```
sudo svcadm enable diffusion
```

To stop the server

```
sudo svcadm disable diffusion
```

Using Maven to build Java Diffusion applications

Apache Maven is a popular Java build tool and is well supported by Java IDEs. You can use Apache Maven to build your Diffusion applications.

The Push Technology public Maven repository

Push Technology publishes Diffusion components and related artifacts to a public Maven repository at the following location: http://download.pushtechnology.com/maven.

The published artifacts include the following:

Table 102: Artifacts

Artifact	Maven coordinates	Description
Diffusion Unified API	`com.pushtechnology.diffusion:diffusion-api:jar:5.1.0`	The Diffusion Unified API interfaces only. Use this artifact for compilation only. The JAR includes the source and Javadoc attachments.
Diffusion Clients	`com.pushtechnology.diffusion:diffusion-client:jar:5.1.0`	The Diffusion client library. The runtime library includes both the Unified API and the Classic API.
mvndar	`com.pushtechnology.tools:dar-maven-plugin:maven-plugin:1.2`	A Maven plugin for building DAR files.

To use the Push Technology public Maven repository, add the following repository description to your `pom.xml` file:

```
<repositories>
    <repository>
        <id>push-repository</id>
        <url>http://download.pushtechnology.com/maven/</url>
    </repository>
</repositories>
```

Related Links

Java on page 258
The Java API comprises a number of classes subordinate to the `com.pushtechnology.diffusion.client` package.

Getting started with the Java API on page 260
The example demonstrates an empty Java client that you can base your clients on.

Building a DAR file on page 384
You can use the Maven™ plugin *mvndar* to build and deploy your publisher DAR file. Alternatively, you can use an Ant build file to perform the same function.

Building the demos using mvndar on page 573
You can use the Maven plugin *mvndar* with the provided `pom.xml` file to build the demos.

Building client applications with Maven

You can build and run Diffusion Java client applications without installing the Diffusion product. The Diffusion client JAR is all you need.

The following `pom.xml` shows how to declare the appropriate dependencies:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://
maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
```

```
    <groupId>com.example</groupId>
    <artifactId>myclient</artifactId>
    <version>1.0-SNAPSHOT</version>

    <repositories>
        <repository>
            <id>push-repository</id>
            <url>http://download.pushtechnology.com/maven/</url>
        </repository>
    </repositories>

    <dependencies>
        <dependency>
            <groupId>com.pushtechnology.diffusion</groupId>
            <artifactId>diffusion-client</artifactId>
            <version>5.1.0</version>
        </dependency>
    </dependencies>
</project>
```

Related Links

Java on page 258
The Java API comprises a number of classes subordinate to the `com.pushtechnology.diffusion.client` package.

Getting started with the Java API on page 260
The example demonstrates an empty Java client that you can base your clients on.

Building publishers and other server application code with Maven

The Diffusion API for publishers and other server application code is not available in the Push Technology public Maven repository. To build server components, you must install the product locally and depend on `diffusion.jar` using a Maven system scope.

The following `pom.xml` shows to declare the dependency on `diffusion.jar`. To use it, you must set the `DIFFUSION_HOME` environment variable to the absolute file path of your Diffusion installation.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.examplecorp</groupId>
    <artifactId>mypublisher</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencyManagement>
        <dependency>
            <groupId>com.pushtechnology.diffusion</groupId>
            <artifactId>diffusion</artifactId>
            <version>local-installation</version>
        </dependency>
    </dependencyManagement>

    <profiles>
        <profile>
            <activation>
                <property>
                    <name>env.DIFFUSION_HOME</name>
                </property>
            </activation>

            <dependencyManagement>
                <dependencies>
```

```
                    <dependency>
                        <groupId>com.pushtechnology.diffusion</groupId>
                        <artifactId>diffusion</artifactId>
                        <version>local-installation</version>
                        <scope>system</scope>
                        <systemPath>${DIFFUSION_HOME}/lib/diffusion.jar</
systemPath>
                    </dependency>
                </dependencies>
            </dependencyManagement>
        </profile>
    </profiles>
</project>
```

mvndar

mvndar is a Maven plugin for creating DAR files. More information about mvndar is available at the following locations:

- http://www.pushtechnology.com/2013/08/13/mvndar-maven-plugin/
- http://pushtechnology.github.io/mvndar/index.html

Example: Using Maven to build the demo applications

If you selected the demo applications when you installed Diffusion, the source files and an example Maven `pom.xml` can be found in the directories beneath the `demos/src` directory. The example uses mvndar, and depends on `diffusion.jar` using system scope.

For more information, see Building the demos using mvndar on page 573.

Related Links

Building a DAR file on page 384
You can use the Maven™ plugin *mvndar* to build and deploy your publisher DAR file. Alternatively, you can use an Ant build file to perform the same function.

Building the demos using mvndar on page 573
You can use the Maven plugin *mvndar* with the provided `pom.xml` file to build the demos.

Chapter 24

Appendix: Protocol

In this section:

- Protocols overview

This appendix describes the protocols for diagnostic purposes and also to allow use of Diffusion from an environment that does not have an API.

Please contact support@pushtechnology.com if you want to make use of the protocol for a client implementation.

The format `0xnn` is used throughout this section to represent hexadecimal byte values. For example, `0x21` represents a byte containing a values of hexadecimal 21 (decimal 33).

Protocols overview

The protocols provided by Diffusion are DPT, WebSockets, HTTP Full Duplex, HTTP Chunked Streaming, HTTP XHR, HTTP iFrame and secure versions of all of these protocols.

The DPT protocol (Diffusion protocol over TCP) creates a TCP connection and uses it to send and receive messages in a full duplex way. Load balancers treat these connections as TCP connections.

The WebSocket implementation provides a browser-based full duplex connection, built on top of WebSocket framing. This complies with the WebSockets standards and is usable with any load balancer or proxy with support for WebSockets.

HTTP Full Duplex acts like HTTP in the initial connection handshake and acts like DPT for the exchange of messages. Messages are exchanged by the client and the server without using HTTP request/response and load balancers or HTTP proxies are not recommended.

HTTP Chunked Streaming provides a streaming connection for messages from the server by using HTTP chunked encoding. A separate TCP connection is used to send messages from the client to the server. This provides two simplex connections, one based on request/response (upstream) and the other streaming data (downstream). This complies with the HTTP 1.1 standard. Load balancers and HTTP 1.1 proxies can cope if you configure them to permit chucked encoding and ignore custom headers. HTTP chunked streaming has two types: iFrame streaming and Comet.

HTTP polling has two types: XHR and iFrame. The XHR connection uses additional and custom headers to encode the details of the request. A poll request remains open until it receives a message. The iFrame uses query string parameters to encode the details of the request. These provide two simplex connections both based on request/response. This complies with the HTTP 1.1 standard. Load balancers and HTTP 1.1 proxies can cope if you configure them to ignore custom headers.

Related Links

Protocols supported on page 115
Each client supports varying transports. A table of the supported transports for each client is presented here.

DPT

Diffusion protocol over TCP

Special byte values

Diffusion uses byte based protocols and certain byte values have special meanings. The following representations are used throughout to signify these values.

	Value	Description
MD	0x00	Message delimiter (Used on for connection)
RD	0x01	Record delimiter
FD	0x02	Field delimiter

Connection protocol

How connections are made to Diffusion components over TCP/IP

All connections involve the initiator of the connection sending a connection request and receiving a connection response.

A connection request has the following format:

```
PVTCcredentials<RD>data<md>
```

Where:

P	Protocol byte – Fixed value of `0x23`
V	Protocol version number as a byte. Current version=4
T	The connection type as a byte
C	Specifies the connection capabilities as a byte.
credentials	Specifies optional authentication credentials. These are specified as username and password tokens. For example `username<FD>password` This record must be provided only if credential information is required by the server. If no credentials are required, only the data can be sent (that is, no `<RD>` is present). Credentials are always UTF-8 encoded.
data	The data can be one or more fields separated by `<FD>`. Data varies according to connection type. Data is always UTF-8 encoded.

A connection response has the following format:

```
PVRM data <MD>
```

Where:

P	Protocol byte – Fixed value of `#` (`0x23`)
V	Protocol version number as a byte. Current Version=4
R	The connection response as a byte
`M`	Message length size as a byte. This indicates the number of bytes allocated to the message size field in a message. Can have a value of 4, 2 or 1. (See message formats).
data	The optional response data can be one or more fields (separated by `<FD>`) and varies according to connection type. Data is always UTF-8 encoded.

Connection types and data

The following table shows non-client types of connection

Type	Connection type	Connection data	Response data
Event publisher	01 (`0x01`)		*identifier*

Notes:

1. *identifier* is a unique string identifier generated by the server.
2. *publisherName* is a unique name for the external publisher.
3. *topicSet* is a comma separated list of valid Topic names to register.
4. *notifications* indicates the notifications that the external publisher receives. This is a list of options separated by "&" from the following list of possibilities:

   - ClientConnected
   - ClientResolved
   - ClientUnsubscribed
   - ClientClosed
   - ClientInvalidSubscription
   - ClientQueueThresholds

   These options are not case sensitive.

Client types

Possible client types (in the range 16 to 63) are as follows:

Type	Connection type
16 (0x10)	Publisher (Client is a publisher at another server)
20 (0x14)	Default type (for user use)
21 (0x15)	Java (API)
22 (0x16)	.NET (API)
23 (0x17)	Flash (JS Bridge)
24 (0x18)	Silverlight (JS Bridge)
25 (0x19)	iOS
27 (0x1B)	Android
28 (0x1C)	BlackBerry®
29 (0x1D)	C client
30 (0x1E)	Perl client
31 (0x1F)	Introspector client
32 (0x20)	Windows Phone client
33 (0x21)	iPad client
34 (0x22)	Flash (Plugin)
35 (0x23)	Silverlight (Plugin)

The connection and response data for all client types is as follows:

Connection data	Response data
topicSet<FD>*client*	*client*

Notes:

- *topicSet* is a comma separated list of one or more topic specifiers indicating the topics to subscribe to on connection. It can be empty to indicate no topics.
- *client* is a unique string identifier for a client generated by the server. It is returned in a client connection response and may optionally be supplied to a client connection request to request a connection to a previously established client session.

Flash policy requests

A connector that supports Flash plugin clients may also service policy file requests. In this case, the following is received:

```
<policy-file-request/>
```

What is returned is a policy file (no response code). No permanent connection is established.

Response codes

100 (0x64)	OK – Connection Successful
101 (0x65)	Invalid connection protocol
103 (0x67)	One or more of the specified topics are invalid
105 (0x69)	Reconnection Successful

110 (0x6E)	Topic already exists (External publisher)
111 (0x6F)	Connection Rejected
127 (0x7F)	Undefined error

Capabilities

The capabilities mask is used by the client when it connects to Diffusion server to indicate to the server what capabilities the client has.

The capabilities mask is a mask of one or more of the following values:

0x01	Can handle encrypted data messages
0x02	Can handle compressed data messages
0x04	Can handle base 64 encoded messages
0x08	Can handle individual features

A value of zero or eight indicates that encoded messages cannot be handled and the server passes all messages in unencoded form. A value of seven or fifteen indicates that all byte encodings can be handled and the server can pass encoded messages. When using DPT this is typically zero or eight unless one or more of the given encodings can be handled for messages passed from the server.

The individual features capability indicates that the client exposes the Unified API.

Messages protocol

The format of messages that are sent over connections and the permitted message types for each connection type.

Protocol

All Diffusion messages are byte based and comprise a fixed header portion, followed by optional header values, followed by an optional data body.

The data portion of a message is optionally encoded with one of a set of permissible byte encodings. These byte encodings are provided for message compression and encryption.

Character encoding of Strings within the data portion of a message is handled by the APIs but if users use the raw protocol they must perform any required character encoding as appropriate. In general it is recommended that characters are encoded using UTF-8 in the data. Where UTF-8 is not used, the message recipient must decode characters as appropriate.

Headers are always UTF-8 encoded.

Message formats

All messages take the form: `LLLLTE{header<RD>}{data}` where:

`LLLL`	This is the total message length (LLLLTE, headers and data). Depending upon the system wide configured size of the length portion of a message, this can comprise 1, 2 or 4 bytes as follows: L for a 8-bit signed two's complement integer allowing for a maximum message length of 127 bytes. LL for a 16-bit signed two's complement integer allowing for a maximum message length of 32,767 bytes. LLLL for a 32-bit signed two's complement integer allowing for a maximum message length of 2,147,483,647 bytes. The value in use is returned on connection.

`T`	Message type byte.
`E`	Byte encoding byte. If there is no data, this is zero.
`{header <RD>}`	Message headers terminated by a record delimiter. Fixed headers will appear first followed by any user defined headers. The fixed headers vary according to message type. Where there is more than one header they are separated by <FD> delimiters. Some message types might have no fixed headers but user headers can be used on any message type. Messages with no fixed headers and no data do not require the <RD> delimiter.
`{data}`	The message data (possibly encoded according to the encoding byte). Some message types may have no message data.

Byte encoding values

0 (`0x00`)	NONE – no encoding applied or requested
1 (`0x01`)	Encryption requested
2 (`0x02`)	Compression requested
3 (`0x03`)	Base-64 encoding requested
17 (`0x11`)	Encrypted
18 (`0x12`)	Compressed
19 (`0x13`)	Base 64 Encoded

In requested cases the message is not encoded but must be encoded by the server before being passed on to a recipient that has sufficient capabilities.

Message types

The following table defines all permitted message types.

Table 103: Message types

Type	Description	Fixed headers (see Key below)	Data
20 (`0x14`)	Topic load message	topic (or topicAlias)	Y
21 (`0x15`)	Delta message	topic	Y
22 (`0x16`)	Subscribe/Register topic or topics	topicSet	N
23 (0x17)	Unsubscribe/ Unregister topic or topics	topicSet	N
24 (`0x18`)	Ping server	timestamp, queueSize	N
25 (`0x19`)	Ping client	timestamp	N
26 (`0x1A`)	Credentials	username, password	N
27 (`0x1B`)	Credentials rejected	username, password	N

Type	Description	Fixed headers (see Key below)	Data
28 (0x1C)	Abort notification		N
29 (0x1D)	Close request		N
30 (0x1E)	Topic load – ACK Required	topic, ackId	Y
31 (0x1F)	Delta – ACK Required	topic, ackId	Y
32 (0x20)	ACK – acknowledge	ackId	N
33 (0x21)	Fetch	topic (and optional user headers)	N
34 (0x22)	Fetch reply	topic (and user headers from request if supplied)	Y
35 (0x23)	Topic status notification	topic (or topicAlias), topicStatus	N
36 (0x24)	Command message	topic, command (, parameters)	Y
40 (0x28)	Command topic load	topic, commandTopicCategory, commandTopicType	Y
41 (0x29)	Command topic notification	topic, notificationType (, parameters)	Y
48 (0x30)	Cancel fragmented message set	correlation id	N

Several message types can also be subject to fragmentation. In this case, the message type is bit-wise OR'd with 0x40. Currently, the types Topic Load, Delta and Fetch Reply can be fragmented, leading to the following message types:

84 (0x54)	Topic Load – Fragmented	topic, correlation id, fragment number(/total fragments), (total size)	Y
85 (0x55)	Delta – Fragmented	topic, correlation id, fragment number(/total fragments), (total size)	Y
98 (0x62)	Fetch Reply – Fragmented	topic, correlation id, fragment number(/total fragments), (total size)	Y

Table 104: Key

Term	Description
ackId	A unique message acknowledgment ID – When client or server receives a type 30 or 31 it must respond with a type 32 with the same ackId.
command (, parameters)	Used on a command topic (that is, the topic that uses command topic data) and is ignored for any other topic. The command is a topic dependent string and the number of parameters is variable.
commandTopicCategory	Can have a value of 0 for user or a specific category number that identifies a Diffusion type category. See Command protocols for Diffusion implementations.

Term	Description
	commandTopicType is a subType which can be used in any way appropriate to the category. Command topic loads can also contain data according to category and type.
notificationType (, parameters)	A reply or notification received on a command topic. The type is a topic dependent string and the number of parameters varies by type.
timestamp	A decimal string representation of the number of milliseconds between the current time and midnight, January 1, 1970 UTC.
topic	A single topic name. **Note:** Where topic aliases are in use then all messages from the server use the alias (including prefix) rather than topic name and all messages to the server can use either topic name or alias.
topicAlias	An optional value passed on a topic load when topic aliasing is configured at the server. It provides an alternative short name and is prefixed by the special alias character (! or 0x21) which can be used in all subsequent message types that must reference the topic. Where an alias has been provided the client must expect all future messages received for the topic to contain only the topic alias.
topicSet	One or more topic names (not aliases) separated by commas (for example,. A/B,C/D). For subscribe/unsubscribe and topic notification request each entry can be a topic selector.
topicStatus	Indicates a change in status of a topic that the client was subscribed to. Currently only a value of R, meaning topic removed is implemented.
total fragments, total size	Used on the first message in a fragmented message set.
queueSize	the size of the client message queue – zero when part of a ping request.
username/password	authentication credentials as passed on connection but can be used to supply or replace credentials after connection.

Message protocols

The following table defines the permitted message types for each connection type. Assume that any message might have a fragmented counterpart.

Table 105: Message interactions

	To server	From server
Client (All types)	21 (Send message) 22 (Subscribe) 23 (Unsubscribe) 24 (Ping server) 25 (Ping client response) 26 (Credentials) 29 (Close notification) 31 (Send message – Ack required) 32 (ACK)	20 (Topic load) 21 (Delta) 24 (Ping server response) 25 (Ping client) 27 (Credentials rejected) 28 (Abort notification) 30 (Topic load – ACK required) 31 (Delta – ACK required) 32 (ACK)

	To server	From server
	33 (Fetch request) 36 (Command)	34 (Fetch reply) 35 (Topic status) 36 (Command) 40 (Topic load – Paged) 41 (Delta – Paged) 42 (Cancel fragmented message set)
Event publisher	20 (Topic load) 21 (Delta) 29 (Close notification) 36 (Command)	21 (Delta) 29 (Close request) 36 (Command)

Fragmented messages protocol

When a message is split into fragments, each fragment in the set is assigned the same correlation ID which is transmitted as the first header after the topic name. The first message of a set will also set a header indicating that this is fragment 1 of *n*, where *n* is the total number of fragments which make up this message. Subsequent fragments provide the fragment number in the set. The first fragment will also provide a header indicating the total length of the data once all fragments have been recombined. For example, for a 250-byte message with a fragment size of 100:

Fragment	Headers
First fragment	topic name correlation ID 1/3 250 {100 bytes of data}
Second fragment	topic name correlation ID 2 {100 bytes of data}
Third fragment	topic name correlation ID 3 {50 bytes of data}

On receipt of a Cancel Fragmented Message set message (type 42), the client expects that no further messages in the set are received and must discard any fragments with the correlation ID that is passed.

HTTP protocol

As well as the standard socket-based protocol, clients can also connect over HTTP protocol.

Protocol

HTTP connections can be:

- Standard HTTP (POST)
- Comet HTTP (POST)
- Iframe (GET)
- Full Duplex over HTTP

- Iframe streaming

The connection protocol is slightly different from the DPT in that connection parameters and connection responses (with the exception of Full Duplex) are passed in HTTP headers or IFrame parameters.

The message protocol is similar to the DPT except that headers are passed as HTTP headers (except Full Duplex) or iframe parameters.

Headers/Parameters

m	Method: HTTP Command (See values below)
ty	type: Identifies the client type (See values below).
ca	Capabilities
v	Version: The protocol version – passed on connection and connection response.
username	Credentials user name. (Specified on connection – optional).
password	Credentials password (Specified on connection if username specified).
c	Client ID: Client Identifier – returned on connection (not Duplex)
d	Data: – This forms the message data body.
t	Topic or topics: Optionally on connection and for topic related commands.
u	User Header: (for messages)
tt	Transport Timeout: (Specified on connection – optional)
s	Sequence: This is the message sequence number. An incrementing sequence number must be sent with each message other than connect and close.
a	Ack ID: This ack ID token when a client send a message where an acknowledgment is required.

Method Value

The following are the possible values of the method parameter or header and is equivalent to a message type. The values in bold are the same as the corresponding message type values. Others are HTTP connection specific.

Value	Command Meaning
0	Connect/Register
1	Poll for messages
2	Send – message from client
3	Duplex – Create Duplex connection
5	SendBatch – Send batched messages from client (Only for HTTPC)
20	Topic load
21	Delta
22	Subscribe
23	Unsubscribe
24	Ping server
25	Ping client
26	Credentials

27	Credentials rejected
28	Abort client
29	Close client
30	Load ack
31	Delta ack
32	Ack
33	Fetch
34	Fetch reply
35	Topic status
36	Command
40	Command load
41	Command notification
48	Cancel fragmented message

Client types

Possible HTTP client types are as follows. The character value is used as the actual `type` header value.

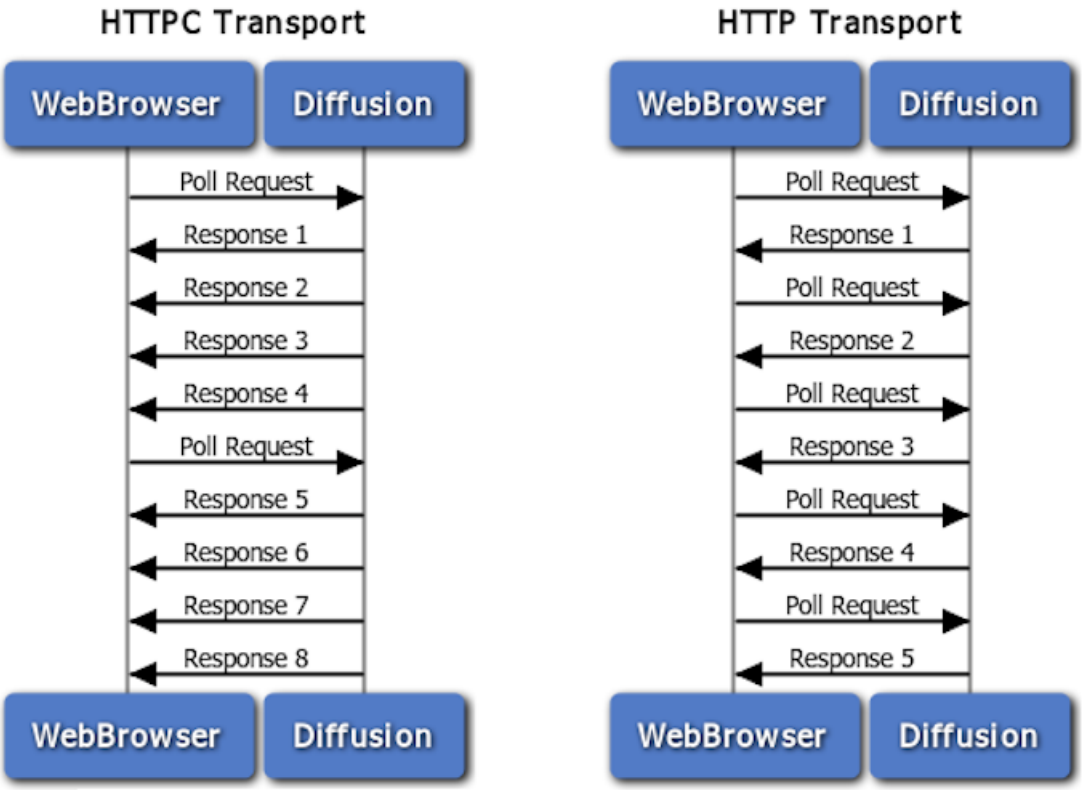
B	Browser default (Callback)
J	Java (API)
F	Flash (Bridge)
FA	Flash (Plugin)
N	.NET
S	Silverlight (Bridge)
SA	Silverlight (Plugin)
W	Windows Phone
U	User
WJ	WebSocket Java
WB	WebSocket Browser
WIOS	WebSocket iOS
WN	WebSocket .NET
CF	Comet Flash (Bridge)
CS	Comet Silverlight (Bridge)
CA	Comet Flash(Plugin)
CN	Comet Silverlight (Plugin)
I	Introspector
WI	WebSocket Introspector

HTTP Comet (HTTPC)

This transport uses long-life HTTP requests and chunked encoding transfer mode as opposed to occasional polling of the server. Instead the server has an open line of communication over which it can push data to the client.

HTTPC can deliver data to the client at any time, not only in response to user input. The data is delivered over a single, previously-opened connection. This approach reduces the latency for data delivery significantly.

The HTTPC request from the client is a polling request that creates a connection to the server. The server can send messages to the client at any time so long as the connection is open. The server closes the connection after it has sent 30Kb of data (this amount can be configured using the comet-bytes-before-new-poll configuration item, see the WebServer client service configuration section). The client then makes another poll request. All messages sent to the client are sent in response to these poll requests. Messages sent from the client to the server will be sent using POST requests. The message formats are exactly the same as for an HTTP based connection.



The HTTPC connection is closed after 30Kb to make it interact well with anti-virus software. Some anti-virus software holds an HTTPC connection until all the data has been sent through. This can cause batching of messages sent over HTTPC.

The client libraries must support the browser to implement HTTPC. For more information about browser support, see Browsers supported on page 118

Full duplex

This protocol allows for a full duplex connection over HTTP.

After a connection is made the message formats are exactly the same as for a socket-based connection, however the data is sent in HTTP chunked encoding format.

A connection request might take the following form:

```
POST /diffusion/ HTTP/1.1
User-Agent: DiffusionClient
Host: localhost:8080
```

```
Transfer-Encoding: chunked
m: 3
ty: U
v: 2
ca: 7
t: Echo
```

The connection response indicates success or failure in the standard HTTP manner but the HTTP headers is followed by a connection response buffer of exactly the same format as DPT and normal message protocol interactions commence.

Iframe streaming

This transport uses an iframe to receive data at any time, over a single, previously opened connection. It is supported only by JavaScript clients.

Iframe streaming works by the following process:

1. The client uses an iframe to create an HTTP request
2. The connection is left open to enable elements to be sent
3. Elements are sent one at a time, wrapped in script tags
4. The iframe loads the script tags and the elements they contain as they are received

Iframe streaming reduces the latency for data delivery significantly, because there is no requirement for multiple request messages to be sent to the server. The server can continue to push data to the client through the existing open connection.

The iframe streaming connection is closed and reopened periodically to interact well with browsers and anti-virus software. Some browsers hold a connection in memory. Closing and reopening the iframe streaming connection avoids causing memory leaks in these browsers. Some anti-virus holds a connection until all the data has been sent through. This can cause batching of messages sent by iframe streaming.

A known limitation of iframe streaming occurs when a client connection is closed. This closure can sometimes result in an `IOException`.

When using iframe streaming, the client uses XHR to communicate with the server. Where XHR is not supported, iframe streaming is not supported.

The client libraries must support the browser to implement iframe streaming. For more information about browser support, see Browsers supported on page 118

WebSocket protocol

Diffusion has support for accepting connections from current and older versions of the WebSocket protocol including hixie-76. The description of the connection protocol is based on RFC 6455.

Protocol

WebSockets are used to provide full duplex communication over port 80, which encourages a firewall-friendly connection. They are possible over other ports.

WebSockets begin with an HTTP-style handshake. It then upgrades to the WebSocket connection. It streams data in a different format to DPT, HTTPC or HTTP. The WebSocket protocol transmits data over the wire in frames. Frames can be sent at any time by both the client and server once the handshake has been completed and before the connection has been closed. Diffusion messages can be split over multiple frames. Once the handshake has been completed, the server must send the client ID.

Connection protocol

To connect using WebSockets the client must make a request to Diffusion with the headers `Connection: Upgrade` and `Upgrade: WebSockets`. This identifies it as an opening of a WebSocket connection. The headers Sec-WebSocket-Key and Sec-WebSocket-Version with the appropriate values must also be included as defined in RFC 6455. The information specific to opening a Diffusion connection are included in the query string of the HTTP request.

ty	type: Identifies the client type either WB (for WebSockets from browser), WJ (for WebSockets from Java), WN (for WebSockets from .NET), WIOS (for WebSockets from iOS), or WI (for WebSockets from the Introspector).
v	Version: The protocol version – passed on connection and connection response.
username	Credentials user name. (Specified on connection – optional).
password	Credentials password (Specified on connection if username specified).
t	Topic or topics: Optionally on connection and for topic related commands.

An example of a WebSocket connection request from the Drawingboard demo is provided below:

```
GET ws://demo.pushtechnology.com/diffusion?
t=drawingboard&v=4&ty=WB&username=default&password= HTTP/1.1
Origin: http://demo.pushtechnology.com
Host: demo.pushtechnology.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: hVlas/+vGiV3uVuG2rs5Rg==
Sec-WebSocket-Version: 13
```

Diffusion responds with an HTTP response which will include the Sec-WebSocket-Accept header to show that the WebSocket connection has been established. An example response follows.

```
HTTP/1.1 101 Switching protocols
Connection: Upgrade
Sec-WebSocket-Accept: 91NYcNxLJPwOylr8mlvB4NoEqDU=
Upgrade: WebSocket
```

This completes a successful WebSocket connection but the connection to Diffusion is not yet finished. The server must send through the connection response before it can send or receive messages. The connection response is sent through from the server to the client over the WebSocket. It is sent as a text frame beginning with the Diffusion protocol version, followed by the string “100” for new connections and “105” for reconnections, followed by the client ID, separated by the field delimiter and terminated by the text postfix byte (0xff).

```
4<FD>10[05]<FD><CLIENT ID>0xff
```

Message protocol

A Diffusion message begins with the message-type byte, using the same values as employed in the DPT protocol. The message headers follow, which are terminated by the record delimiter. The payload of the message follows next. The Diffusion server can pack more than one message within a WebSocket message, in which case messages are delimited with character 0x08. Diffusion messages are sent in data frames using the text opcode.

Diffusion reuses the WebSocket message framing, so no explicit size data is required. There is no encoding byte as Diffusion does not support message encodings for WebSockets.

```
T{header<RD>}{data}D
```

Where T is the message-type byte and D is the message separator (0x08).

Related Links

http://tools.ietf.org/html/rfc6455

Command protocols

The message protocol defines command messages which are automatically routed to and processed by topics that have CommandTopicData.

There are several implementations of command topics and the message content for each is documented here. This documentation is provided for diagnostic purposes only and it is not expected that client applications must know these message formats as the Diffusion client APIs must provide appropriate handlers which format and unpack the messages.

Service topic protocol

A service topic is a type of command topic used for request/response type services. Client APIs provide handlers for this protocol.

Topic load (Message type 40)

The commandTopicCategory of the topic load is "0".

The commandTopicType is a service type as defined for the topic data.

The headers and data can be anything set on the topic data.

Commands (Message type 36)

Type	Description	Headers / Data
requestType	The request type can be anything that the service understands.	0 = Request Id This is a unique correlation ID generated by the client and is returned with any response. Further headers and data can be supplied as input to the request.

Notifications (Message type 41)

Type	Description	Headers / Data
responseType	The response type can be anything as defined by the service.	0 = Request Id 1-n = Other user headers (optional) Data = optional user data
!ERR	Error response	0 = Request Id 1 = Error type SERVICE("SRV"), DUPLICATE("DUP"), TIMEOUT("TIM"), INVALID("INV"), USER("USR"); Data is in fields as follows: 0 = Error Message 1 = Exception message (optional) 2 = Further info (optional)

Paged topic protocol

A paged topic is a type of command topic. Client APIs provide handlers for this protocol.

Topic load (Message type 40)

The commandTopicCategory of the topic load is 1.

The commandTopicType might be PS for Paged String data or PR for Paged Record data

At future releases the data body can contain metadata.

Commands (Message type 36)

Type	Description	Headers / Data
O	Open Paging	0 = Lines per page 1 = Start page (-1 for last)
R	Refresh Page	
N	Next Page	
P	Prior Page	
F	First Page	
L	Last Page	
n (a numeric)	Page number	
C	Close Paging	

Notifications (Message type 41)

Type	Description	Headers / Data
P	Page Delivered in response to O,R,N,P,F,L commands.	0 = Current Page Number 1 = Last Page Number 2 = Total Number of Lines Data = lines of data separated according to type.
S	Status Update to status so page range or dirty mark can be refreshed.	0 = Current Page Number 1 = Last Page Number 2 = Total Number of Lines 3 = dirty (true or false)
U	Update A line on the current page has been updated.	0 = Current Page Number 1 = Last Page Number 2 = Total Number of Lines 3 = Index (on page) of line to update Data = single line of data
A	Add Lines have been added to the end of the current page	0 = Current Page Number 1 = Last Page Number 2 = Total Number of Lines Data = lines of data (no more than fill the page).

Notification topic protocol

The usage of command message protocol by topic notification topic data

This specification defines the use of version 2 (available at Diffusion release 4.6) of the notification topic protocol.

A 'topic notification topic' is a type of command topic. Client APIs can provide handlers for this protocol.

Topic load (Message type 40)

The commandTopicCategory of the topic load is `3`.

The commandTopicType is set to `TN`

Header 0 contains the protocol version number (currently `2`)

The usage of the body is to be defined

Commands (Message type 36)

Table 106: Commands (Message type 36)

Type	Description	Headers / Data
`S`	Select/Amend notifications	H0 = Mode – `R`=Replace,`A`=Append,`D`=Remove,`X`=Clear H1 = TopicSet (comma separated selectors)
`L`	Set detail level	H0 = Level `0` = Minimum (only type) – this is the default `1` = Properties only `2` = Metadata only `3` = All `4` = None – special case if just deletions and/or updates required H1 = Metadata Caching (`true` or `false`) H2 = Notify Deletions (`true` or `false`) – false if not present H3 = Notify Updates (`true` or `false`) – false if not present

Table 107: Notifications (Message type 41)

Type	Description	Headers / Data
`A`	Topic Added	H0 = Topic Name data:- (3 records – if not selected by level the record is present but empty) R0 Basic info F0=Topic type(see below) R1 Topic Properties key/value field pairs R2 Metadata F0=Metadata type `X`=XML,`R`=Reference(name), `N`=None F1=Metadata reference (type X and R only) F2=Metadata content (type X only)
`D`	Topic Removed	H0 = Topic Name
`U`	Topic Properties Changed	R0 Topic Properties key/value field pairs

Table 108: Topic types

Code	Type	Metadata type
N	Simple	
S	Single value	MField
R	Record	MMessage
G	Protocol buffers	optional any
U	Custom	optional any
SD	Slave	from target
SV	Service	optional any
PS	Paged String	
PR	Paged record	MRecord
TN	Topic notification	
RO	Routing	
C	Child list	

Table 109: Topic properties

Key	Value	Type	Topic types	Optional/Mandatory & default
ATC	Attachment class name	String	all	Optional – default no attachment
LOC	Lockable	Boolean	all	Optional – default true
REF	Reference	String	all	Optional – default no reference
SHC	`TopicSubscriptionHandler` class name	String	all	Optional – default no handler
FHC	`TopicFetchHandler` class name	String	all	Optional – default no handler
DMC	Delta message capacity default	Integer	all	Optional – default -1
LMC	Load message capacity default	Integer	all	Optional – default -1
ISM	Introspection mask	Byte	all	Optional – default 0
<O	Lock timeout	Long	all	Optional – default 5000
TOU	Tidy on unsubscribe	Boolean	all	Optional – default false
ALI	Aliasing	Boolean	all	Optional – default as publisher
ASU	Auto subscribe	Boolean	all	Optional – default false
LEN	Load encoding	Byte	all	Optional – default 0
DEN	Delta encoding	Byte	all	Optional – default 0
LFS	Load fragment size	Integer	all	Optional – default no fragments

Key	Value	Type	Topic types	Optional/Mandatory & default
DFS	Delta fragment size	Integer	all	Optional – default no fragments
LFL	Load fragment lifecycle (S=n,M=n,P=n)	String	all	Optional – default no lifecycle
DFL	Delta fragment lifecycle (S=n,M=n,P=n)	String	all	Optional – default no lifecycle
DIC	TopicDataInitialiser class name	String	pub	Optional – default no initializer
LHD	Load headers (field delimited list)	List	pub	Optional – default no load headers
LAR	Load ack required	Boolean	pub	Optional – default no ack required
DAR	Delta ack required	Boolean	pub	Optional – default no ack required
G_PC	Proto class name	String	G	Mandatory for type
G_MN	PB message name	String	G	Mandatory for type
G_UM	PB update mode – (P) Partial or (F) Full	String	G	Optional – default P
G_DV	PB deletion value	String	G	Optional – default <DEL>
U_HC	CustomTopicDataHandler class name	String	U	Mandatory for type
R_EFV	Empty field value	String	R	Optional – default = ""
PX_CC	Paged Comparator Class	String	PR,PS	Optional – default unordered
PX_DP	Paged duplicates policy – (N) Not allowed, (F) First or (L) Last	String	PR,PS	Optional – Default is N
SV_TY	Service type	String	SV	Mandatory for type
SV_HC	ServiceHandler class name	String	SV	Mandatory for type
SV_TO	Service request timeout	Long	SV	Optional – default 5000
SV_TT	Service target topic	String	SV	Optional – default parent is used
SV_HD	Service header options	List	SV	Optional – default no options
SD_MT	Slave master topic	String	SD	Mandatory for type
RO_HC	RoutingTopicDataSubscriptionHandler	String	RO	Mandatory for type
TN_MC	Notify – metadata caching	Boolean	TN	Optional – default false
TN_NA	Notify – notify all topics	Boolean	TN	Optional – default false

The entry "pub" refers to publishing topics – for example, U,G,R,S,SD

"List" type is a list of Strings delimited by list delimiter byte (0x06).

Appendix

Appendices

In this section:

- Document conventions
- Glossary
- Trademarks
- Copyright Notices

Appendix A

Document conventions

This user manual uses certain typographic conventions to distinguish between different types of information.

The following table describes how different types of information are represented typographically.

Table 110: Typographic conventions used in this manual

Convention	Usage
`Monospace`	Indicates the following items: • Source code • Class or method names • Command names • File paths • Information input by the user
Bold	Indicates the following items: • Interface element titles (buttons, menu items, field names) • Window or panel titles
Italic	Indicates the following items: • New terms – when appearing in the text • Variable values – when appearing in code or syntax examples
Greater than sign (>)	Indicates a menu item or sequence of menu items. For example, "Choose **File** > **Save**" means choose the **Save** item from the **File** menu.

Appendix

B

Glossary

In this section:

- A
- C
- D
- E
- F
- H
- I
- L
- M
- P
- Q
- R
- S
- T
- U

The glossary contains key terms associated with Diffusion and their definitions.

A

acceptor

A thread that accepts an open connection from a client.Too few acceptors can cause client connections to time out. Too many acceptors can cause resource drains on the system.

acknowledgment

A message sent by the recipient of a message or update to inform the sender of that message or update that it was received.Not all messages or updates are acknowledged. A message of update must specify that acknowledgment is required to receive an acknowledgment.

ack

application programming interface

A set of contracts that you can program against to interact with Diffusion

The following APIs are available:

- Publisher API
- Unified API
- Classic API

API

C

callback

In the Unified API, a callback is an object, specific to a single call, that is used to respond to a request.

client

An entity that connects to the server and subscribes to topics.

Typically a client is a user-written application communicating with a Diffusion server through a client API or the Diffusion protocol. A publisher can be a client of another publisher in a distributed environment.

client library

A library that is included in a client application to enable interaction with the server.

conflation

The facility to prevent transmission of unnecessary duplicate message information to reduce network traffic. Conflation removes outdated information from the queue of content to be sent and either replaces the outdated information with the updated information or appends the updated information to the end of the queue.

connector

A configured point of connection to a server.

There can be one or more connectors (each listening on a different port). Each connector can accept single or multiple types of connection.

consume

When a message or update is received by a topic listener and that listener chooses not to pass on the message or update to subsequent topic listeners. Topic streams cannot choose to consume messages or updates they receive.

control client

A client that can perform control actions.

A control client registers handlers with the server at server or topic level.

credentials

A piece of information that is used to authenticate a principal.

D

DAR file

A Diffusion archive file. This file contains a publisher and can be deployed on the server.

delimiter

A byte value that is used as a separator in messages.

Depending on the type of message, it can contain field delimiters or record delimiters.

delta

A message that is sent to a client subscribed to a topic. The delta contains only information that has been updated since the last message was sent to the client.

Diffusion

The Diffusion product comprising the Diffusion server and client libraries.

dirty

The state of a page in a paged topic at the client if its content has been changed on the server, but not updated on the client.

DPT

A Diffusion proprietary streaming protocol that clients can use to communicate with the server.

Diffusion protocol for TCP

E

event publisher

An event publisher is an application connecting to a Diffusion server using the event publisher protocol to send messages to the server.

Each message is routed by the server to the publisher owning the message's topic. The event publisher can be an API or a raw protocol connection. The purpose of an event publisher is to act as a data feed to publishers.

F

feature

An API module that contains a conceptual set of facilities.

fetch

A request from a client to a publisher for the current state of a topic.

field

A section of content that contains data of a specific type. Fields are nested inside records. A record can contain one or many fields.

H

handler

In the Unified API, a handler is an object responsible for responding to one or more instances of a single type of request.

I

initial topic load

The message sent to a client when it first subscribes to a topic. This message contains the current state of all data related to the topic.

ITL

L

line

An entry in a page of a paged topic.The number of lines on a page is specified by the client that views the paged topic.

listener

In the Unified API, a listener is an object that is always called when a particular event occurs.

M

message

A message is a series of bytes of information formatted according to the Diffusion protocol which can be sent between components utilizing Diffusion.

message queue

A queue of messages. Each client connection to Diffusion has such a queue at the server upon which messages are put for sending to the client.

queue

metadata

Data about data. In Diffusion metadata is used to define the structure of messages.

message metadata

multicast

To send data to several recipients at the same time.

The datagrid uses multicasting to locate other datagrid nodes.

P

paged topic

A topic whose data is formatted in lines and viewed as pages containing one or more lines.

ping

A query sent by a publisher, client, or Diffusion server to a connected component to check that the connection exists and the latency of the connection.
The following types of ping are available:

server ping
: A client pings the Diffusion server.

client ping
: A publisher pings a specific client.

system ping
: The Diffusion server pings all clients at a regular interval.

principal

An identity that can be authenticated by the Diffusion or a control client.

A principal can be a user or client. After a principal has been authenticated, it can be authorized to access actions or resources.

protocol

A protocol defines the exact format of data passed between server and client.

Where an API is not available for a particular language, socket based connections can be made to a Diffusion server and the raw protocol used for communication.

publisher

The component which publishes messages relating to one or more topics.

A server can host one or more publishers. Messages sent by clients on particular topics are routed to the Publisher that owns the topic. Publisher functionality is provided by users by writing a Java publisher class.

Q

message queue

A queue of messages. Each client connection to Diffusion has such a queue at the server upon which messages are put for sending to the client.

queue

R

record

A section of content that acts as a container for a set of fields. Inside the content of a message or update you can have zero, one, or many records. A record can contain one or many fields.

regular expression

A string that uses special characters to describe a search pattern.

Diffusion uses Java-style regular expressions.

regex

S

server

The component that hosts topics and publishers. A server broadcasts topic messages to all subscribed clients.

Clients can connect to servers through the Unified API or Classic API.

Diffusion server

session

A logical context between a client and one or more Diffusion servers.

Typically, a session represents a single client connection to a single server. However, in the event of connection failure the session can automatically reconnect to the same server or even failover to another server and still retain its context.

structural conflation

A form of conflation that enables you to define the operations performed on outdated content. You can merge, aggregate, reverse or combine the effects of multiple changes into a single consistent and current notification to the client.

stream

In the Unified API, a stream is a sequence of responses to a single request.

subscribe

A client registering interest in a topic such that the client receives messages that are sent to that topic.

T

throttling

Limiting the volume of messages that the server transmits to a client within a specified period of time.

Throttling can be used to limit bandwidth usage or to prevent more messages being sent to a client than the client can handle.

topic

A logical channel through which messages are distributed.

Topics provide a logical link between publishers and clients. Publishers publish messages to topics. Clients subscribe to topics to receive messages published to that topics.

Diffusion topics are bi-directional: a client can send a message to a topic and the topic routes the message to the topic's publisher or publishers.

topic path

A string representation of a location in the topic tree.

A topic path consists of parts separated by a slash character (/).

topic name

hierarchic topic name

full topic name

topic path prefix

The root part of a topic selector.

A concrete topic path to the most specific part of the topic tree that contains all topics that the selector can specify. For example, for the topic selector `?foo/bar/baz/.*/bing`, the topic path prefix is `foo/bar/baz`.

path prefix

topic selector

An object that retrieves one or more topics based on their topic paths.

A topic selector uses a pattern expression, which can include one or more regular expressions, to match to the path of one of more topics.

selector

topic tree

The organization structure of topics at a server.

A topic can have subtopics and can itself be a subtopic of another topic. The topics of all publishers at the server are in the topic tree.

topic hierarchy

transport

An implementation of a network protocol. The mechanism by which clients communicate with the Diffusion server.

U

update

A change to the state of a topic and the data on the topic. Updates to topics are done by publishers.

Appendix C

Trademarks

The following trademarked terms are included in this manual.

Diffusion is trademark of Push Technology Ltd.

ActionScript, Adobe, Flash, and Flexare registered trademarks of Adobe Systems Incorporated.

AIX™, Cast Iron®, and WebSphere are trademarks of IBM.

Amazon and Amazon EC2 are trademarks of Amazon.

Android and Chrome are trademarks of Google Inc.

Ant, Apache, Apache Derby™, Apache Tomcat™, and Maven are trademarks of The Apache Software Foundation.

BlackBerry is a registered trademark of RIM.

CentOS and Red Hat are trademarks or registered trademarks of Red Hat, Inc.

Dell is trademark of Dell, Inc.

Eclipse is a trademark of the Eclipse Foundation, Inc.

F5 is a registered trademark of F5 Networks, Inc.

Firefox is a registered trademark of Mozilla Foundation.

Hazelcast is a trademark of Hazelcast Inc.

Intel and Xeon are trademarks of Intel Corporation.

Internet Explorer, Microsoft, Silverlight, and Windows are trademarks or registered trademarks of Microsoft Corporation.

iOS is a registered trademark of Cisco.

Java, JavaScript, Oracle, and Solaris™ are trademarks or registered trademarks of Oracle Corporation.

Joyent is a trademarks of Joyent.

Linux is a trademark of Linus Torvalds.

Mac®, OS X, Safari, and Siri® are registered trademarks of Apple Inc.

Nagios® is a registered trademark of Nagios Enterprises.

Opera is a registered trademark of Opera Software ASA.

Splunk is a trademark of Splunk, Inc.

SUSE is a registered trademark of SUSE LLC.

TIBCO Enterprise Message Service is a trademark of TIBCO Software Inc.

Ubuntu is a registered trademark of Canonical Ltd.

UNIX is a registered trademark of The Open Group.

VeriSign® is a registered trademark of VeriSign, Inc.

VMware and VMware vSphere are registered trademarks of VMware, Inc.

Zendesk®is a registered trademark of Zendesk, Inc.

Appendix D

Copyright Notices

In this section:

- Apache Commons Codec
- CocoaAsyncSocket
- cron4j
- d3
- FastColoredTextBox
- Fluent validation
- GeoIP API
- GeoLite City Database
- geronimo-jms_1.1_spec
- Hazelcast
- HPPC
- htmlcompressor
- IKVM
- JCIP Annotations
- JCommon
- jQuery
- JSON
- json-simple
- JZlib
- Knockout
- Metrics
- opencsv
- Picocontainer
- Protocol Buffers
- Rickshaw
- SLF4J
- SocketRocket
- SWT 2D Graphics
- Tapestry (Plastic)
- TrueLicense
- Licenses

Diffusion uses third party, open source software. The rights to this software are not owned by Push Technology and the software is distributed under different licensing agreements. The distribution and use of third-party software is subject to the applicable terms.

The following sections list the software used, their licenses, copyright notices and disclaimers.

Apache Commons Codec

Version 1.6

http://commons.apache.org/codec/

Apache Commons Codec is distributed under the Apache License 2.0.

Copyright 2002-2011 The Apache Software Foundation. All Rights Reserved

Additional notices

The following information is included in the `NOTICE.txt` file that accompanies the source:

```
Apache Commons Codec
Copyright 2002-2011 The Apache Software Foundation

This product includes software developed by
The Apache Software Foundation (http://www.apache.org/).

---------------------------------------------------------------------------
src/test/org/apache/commons/codec/language/DoubleMetaphoneTest.java
 contains
test data from http://aspell.sourceforge.net/test/batch0.tab.

Copyright (C) 2002 Kevin Atkinson (kevina@gnu.org). Verbatim copying
and distribution of this entire article is permitted in any medium,
provided this notice is preserved.
---------------------------------------------------------------------------
```

CocoaAsyncSocket

Version 7.3.4

https://github.com/robbiehanson/CocoaAsyncSocket

The CocoaAsyncSocket project is in the public domain.

The original TCP version (AsyncSocket) was created by Dustin Voss in January 2003.

Updated and maintained by Deusty LLC and the Apple development community.

cron4j

Version 2.2.5

http://www.sauronsoftware.it/projects/cron4j/

Cron4j is distributed under the LGPL 2.1.

Copyright (C) 2007-2010 Carlo Pelliccia (www.sauronsoftware.it)

Source code is available from the following location: http://sourceforge.net/projects/cron4j/files/cron4j/2.2.5/cron4j-2.2.5.zip/download/

For a fee, Push Technology can also provide this source on a CD. To request a copy, contact support@pushtechnology.com.

d3

Version 3.2.1

http://d3js.org/

d3 is distributed under the Apache License 2.0.

Copyright (c) 2010-2014, Michael Bostock

FastColoredTextBox

https://github.com/PavelTorgashov/FastColoredTextBox

FastColoredTextBox distributed under the LGPL 3.0 or later.

Copyright (C) Pavel Torgashov, 2011-2014.

Source code is available at the following location: https://github.com/PavelTorgashov/FastColoredTextBox

For a fee, Push Technology can also provide this source on a CD. To request a copy, contact support@pushtechnology.com.

Fluent validation

Version 3.3.1.0

http://fluentvalidation.codeplex.com/

Fluent validation is distributed under the Apache License 2.0.

Copyright Jeremy Skinner

GeoIP API

Version 1.2.13

http://www.maxmind.com/en/opensource

The GeoIP API is distributed under the LGPL 2.1 or later.

Copyright (C) 2003 MaxMind LLC. All Rights Reserved

GeoLite City Database

http://www.maxmind.com/en/opensource

The GeoLite City Database is distributed under the Creative Commons Attribution-ShareAlike 3.0 Unported License.

Copyright MaxMind LLC

geronimo-jms_1.1_spec

Version 1.1

http://geronimo.apache.org/

geronimo-jms_1.1_spec is distributed under the Apache License 2.0

Copyright 2003-2006 The Apache Software Foundation

Additional notices

The following information is included in the `NOTICE.txt` file that accompanies the source:

```
Apache Geronimo
Copyright 2003-2006 The Apache Software Foundation

This product includes software developed by
The Apache Software Foundation (http://www.apache.org/).
```

Hazelcast

Version 3.2.4

http://www.hazelcast.org/

Hazelcast is distributed under the Apache License 2.0

Copyright (c) 2008-2014, Hazelcast, Inc. All Rights Reserved.

Additional notices

The following information is included in the `NOTICE.txt` file that accompanies the source:

```
**
**  NOTICE file corresponding to the section 4 (d) of the Apache License,
**  Version 2.0, in this case for the Hazelcast distribution.
**

The end-user documentation included with a redistribution, if any,
must include the following acknowledgement:

 "This product includes software developed by the Hazelcast
  Project (http://www.hazelcast.com)."

Alternately, this acknowledgement may appear in the software itself,
if and wherever such third-party acknowledgements normally appear.

The name "Hazelcast" must not be used to endorse or promote products
derived from this software without prior written permission. For
written permission, please contact talip@hazelcast.com or fuad@hazelcast.com

Copyright (c) 2008-2014 Hazelcast Inc.  All rights reserved.
```

HPPC

Version 0.6.0

http://labs.carrotsearch.com/hppc.html

HPPC is distributed under the Apache License 2.0.

Copyright 2010-2013, Carrot Search s.c., Boznicza 11/56, Poznan, Poland

htmlcompressor

Version 1.5.2

http://code.google.com/p/htmlcompressor/

The htmlcompressor is distributed under the Apache License 2.0.

Copyright 2009-2011 Sergiy Kovalchuk

Additional notices: Apache License 2.0 Notice

IKVM

Version: 7.2.4630.5

http://www.ikvm.net/

IKVM is distributed under the GPL v2 with Classpath Exception.

Copyright (C) 2002-2011 Jeroen Frijters

Source code is available at the following location: http://sourceforge.net/projects/ikvm/files/ikvm/7.2.4630.5/

For a fee, Push Technology can also provide this source on a CD. To request a copy, contact support@pushtechnology.com.

JCIP Annotations

Version 1

https://github.com/stephenc/jcip-annotations

jcip-annotations is distributed under the Apache License 2.0

Copyright 2013 Stephen Connolly.

JCommon

Version 1.0.16

http://www.jfree.org/jcommon/

JCommon is distributed under the LGPL 2.1 or later.

Copyright 2000-2006, by Object Refinery Limited and Contributors.

Source code is available at the following location: http://sourceforge.net/projects/jfreechart/files/3.%20JCommon/1.0.16/

For a fee, Push Technology can also provide this source on a CD. To request a copy, contact support@pushtechnology.com.

jQuery

Version: 1.7.1

https://jquery.org/

jQuery is distributed under the MIT License.

Copyright 2014 jQuery Foundation and other contributors http://jquery.com/

JSON

http://www.json.org/java/

JSON is distributed under the JSON License.

Copyright (c) 2002 JSON.org

json-simple

Version 1.1.1

https://code.google.com/p/json-simple/

json-simple is distributed under the Apache License 2.0.

Copyright (c) Yidong Fang, Chris Nokleberg

JZlib

Version 1.02

http://www.jcraft.com/jzlib/

JZlib is distributed under the BSD 3-clause License. This has not been modified, it has been compiled from the source code for distribution.

Copyright 2000-2011 ymnk, JCraft,Inc. All rights reserved.

Knockout

Version 2.1.0

http://knockoutjs.com/

Knockout is distributed under the MIT License.

Copyright (c) Steven Sanderson, the Knockout.js team, and other contributors

Metrics

Version 3.0.0-BETA

http://metrics.codahale.com/

Metrics is distributed under the Apache License 2.0.

Copyright (c) 2010-2013 Coda Hale, Yammer.com

Additional notices

The following information is included in the `NOTICE.txt` file that accompanies the source:

```
Metrics
Copyright 2010-2013 Coda Hale and Yammer, Inc.

This product includes software developed by Coda Hale and Yammer, Inc.

This product includes code derived from the JSR-166 project
 (ThreadLocalRandom, Striped64,
LongAdder), which was released with the following comments:

    Written by Doug Lea with assistance from members of JCP JSR-166
    Expert Group and released to the public domain, as explained at
    http://creativecommons.org/publicdomain/zero/1.0/
```

opencsv

Version 2.3

http://opencsv.sourceforge.net/

opencsv is distributed under the Apache License 2.0.

Copyright 2005 Bytecode Pty Ltd.

Picocontainer

Version 2.14.3

http://picocontainer.codehaus.org/

Picocontainer is distributed under the BSD 3-clause License.

Copyright (c) 2003-2008 PicoContainer Organization. All rights reserved.

Protocol Buffers

Version 2.4.1

http://code.google.com/p/protobuf/

Component: Server

Google's Protocol Buffers are distributed under the BSD 3-clause License.

Copyright 2008 Google Inc. All rights reserved.

Rickshaw

http://code.shutterstock.com/rickshaw/

Rickshaw is distributed under the MIT License.

Copyright (C) 2011-2013 by Shutterstock Images, LLC

SLF4J

Version 1.7.7

http://www.slf4j.org/

SLF4J is distributed under the MIT License.

Copyright (c) 2004-2013 QOS.ch All rights reserved.

SocketRocket

Version 0.3.1-beta2

https://github.com/square/SocketRocket

SocketRocket is distributed under the Apache License 2.0.

Copyright 2012 Square Inc.

SWT 2D Graphics

http://www.jfree.org/jfreechart/

SWT 2D Graphics is distributed as part of JFreeChart under the LGPL 2.1 or later.

Copyright 2000-2008, by Object Refinery Limited and Contributors.

Copyright 2006-2008, by Henry Proudhon and Contributors.

Source code is available at the following location: http://sourceforge.net/projects/jfreechart/files/1.%20JFreeChart/1.0.13/

For a fee, Push Technology can also provide this source on a CD. To request a copy, contact support@pushtechnology.com.

Tapestry (Plastic)

Version 5.3.7

http://tapestry.apache.org/

Tapestry is distributed under the Apache License 2.0.

Copyright 2011, 2012 The Apache Software Foundation

Additional notices

The following information is included in the `NOTICE.txt` file that accompanies the source:

```
This product includes software developed by
The Apache Software Foundation (http://www.apache.org/).

Please refer to the NOTICE.txt in each sub-module to
identify further dependencies.

The Maven central repository is the preferred method to download Tapestry
and its dependencies. The binary archive includes just basic
dependencies for tapestry-core; using other modules (such as
tapestry-hibernate or any of the others) requires downloading
additional dependencies. Please refer to the Maven POM for each module
to identify its dependencies.
```

TrueLicense

Version 1.33

http://truelicense.java.net/

This version of TrueLicense was distributed under the EPL.

Copyright 2005-2012 Schlichtherle IT Services

Licenses

The following licenses are used by the third party, open source software that is distributed with Diffusion.

Apache License 2.0

Apache License

Version 2.0, January 2004

http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the

Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

(a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

(b) You must cause any modified files to carry prominent notices stating that You changed the files; and

(c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied,

including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

Related Links

Apache Commons Codec on page 634

d3 on page 634

Fluent validation on page 635

geronimo-jms_1.1_spec on page 635

Hazelcast on page 635

HPPC on page 636

htmlcompressor on page 636

JCIP Annotations on page 637

json-simple on page 637

Metrics on page 638

opencsv on page 638

SocketRocket on page 639

Tapestry (Plastic) on page 639

BSD 3-clause License

Copyright (c) <YEAR>, <OWNER>

All rights reserved.

> **Note:** The copyright statement above is included in its completed form in the sections of this document specific to the individual products covered by this license.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The names of the authors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL JCRAFT, INC. OR ANY CONTRIBUTORS TO THIS SOFTWARE BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Related Links

Protocol Buffers on page 638

JZlib on page 637

Picocontainer on page 638

Eclipse Public License – v 1.0

THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS OF THIS ECLIPSE PUBLIC LICENSE ("AGREEMENT"). ANY USE, REPRODUCTION OR DISTRIBUTION OF THE PROGRAM CONSTITUTES RECIPIENT'S ACCEPTANCE OF THIS AGREEMENT.

1. DEFINITIONS

"Contribution" means:

- a) in the case of the initial Contributor, the initial code and documentation distributed under this Agreement, and
- b) in the case of each subsequent Contributor:
  - i) changes to the Program, and
  - ii) additions to the Program;

where such changes and/or additions to the Program originate from and are distributed by that particular Contributor. A Contribution 'originates' from a Contributor if it was added to the Program by such Contributor itself or anyone acting on such Contributor's behalf. Contributions do not include additions to the Program which: (i) are separate modules of software distributed in conjunction with the Program under their own license agreement, and (ii) are not derivative works of the Program. "Contributor" means any person or entity that distributes the Program.

"Licensed Patents" mean patent claims licensable by a Contributor which are necessarily infringed by the use or sale of its Contribution alone or when combined with the Program.

"Program" means the Contributions distributed in accordance with this Agreement.

"Recipient" means anyone who receives the Program under this Agreement, including all Contributors.

2. GRANT OF RIGHTS

a) Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free copyright license to reproduce, prepare derivative works of, publicly display, publicly perform, distribute and sublicense the Contribution of such Contributor, if any, and such derivative works, in source code and object code form.

b) Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free patent license under Licensed Patents to make, use, sell, offer to sell, import and otherwise transfer the Contribution of such Contributor, if any, in source code and object code form. This patent license shall apply to the combination of the Contribution and the Program if, at the time the Contribution is added by the Contributor, such addition of the Contribution causes such combination to be covered by the Licensed Patents. The patent license shall not apply to any other combinations which include the Contribution. No hardware per se is licensed hereunder.

c) Recipient understands that although each Contributor grants the licenses to its Contributions set forth herein, no assurances are provided by any Contributor that the Program does not infringe the patent or other intellectual property rights of any other entity. Each Contributor disclaims any liability to Recipient for claims brought by any other entity based on infringement of intellectual property rights or otherwise. As a condition to exercising the rights and licenses granted hereunder, each Recipient hereby assumes sole responsibility to secure any other intellectual property rights needed, if any. For example, if a third party patent license is required to allow Recipient to distribute the Program, it is Recipient's responsibility to acquire that license before distributing the Program.

d) Each Contributor represents that to its knowledge it has sufficient copyright rights in its Contribution, if any, to grant the copyright license set forth in this Agreement.

3. REQUIREMENTS

A Contributor may choose to distribute the Program in object code form under its own license agreement, provided that:

- a) it complies with the terms and conditions of this Agreement; and
- b) its license agreement:
  - i) effectively disclaims on behalf of all Contributors all warranties and conditions, express and implied, including warranties or conditions of title and non-infringement, and implied warranties or conditions of merchantability and fitness for a particular purpose;
  - ii) effectively excludes on behalf of all Contributors all liability for damages, including direct, indirect, special, incidental and consequential damages, such as lost profits;
  - iii) states that any provisions which differ from this Agreement are offered by that Contributor alone and not by any other party; and
  - iv) states that source code for the Program is available from such Contributor, and informs licensees how to obtain it in a reasonable manner on or through a medium customarily used for software exchange.

When the Program is made available in source code form:

- a) it must be made available under this Agreement; and
- b) a copy of this Agreement must be included with each copy of the Program.

Contributors may not remove or alter any copyright notices contained within the Program.

Each Contributor must identify itself as the originator of its Contribution, if any, in a manner that reasonably allows subsequent Recipients to identify the originator of the Contribution.

4. COMMERCIAL DISTRIBUTION

Commercial distributors of software may accept certain responsibilities with respect to end users, business partners and the like. While this license is intended to facilitate the commercial use of the Program, the Contributor who includes the Program in a commercial product offering should do so in a manner which does not create potential liability for other Contributors. Therefore, if a Contributor includes the Program in a commercial product offering, such Contributor ("Commercial Contributor") hereby agrees to defend and indemnify every other Contributor ("Indemnified Contributor") against any losses, damages and costs (collectively "Losses") arising from claims, lawsuits and other legal actions brought by a third party against the Indemnified Contributor to the extent caused by the acts or omissions of such Commercial Contributor in connection with its distribution of the Program in a commercial product offering. The obligations in this section do not apply to any claims or Losses relating to any actual or alleged intellectual property infringement. In order to qualify, an Indemnified Contributor must: a) promptly notify the Commercial Contributor in writing of such claim, and b) allow the Commercial Contributor to control, and cooperate with the Commercial Contributor in, the defense and any related settlement negotiations. The Indemnified Contributor may participate in any such claim at its own expense.

For example, a Contributor might include the Program in a commercial product offering, Product X. That Contributor is then a Commercial Contributor. If that Commercial Contributor then makes performance claims, or offers warranties related to Product X, those performance claims and warranties are such Commercial Contributor's responsibility alone. Under this section, the Commercial Contributor would have to defend claims against the other Contributors related to those performance claims and warranties, and if a court requires any other Contributor to pay any damages as a result, the Commercial Contributor must pay those damages.

5. NO WARRANTY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, THE PROGRAM IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OR CONDITIONS OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Each Recipient is solely responsible for determining the appropriateness of using and distributing the Program and assumes all risks associated with its exercise of rights under this Agreement, including but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or loss of data, programs or equipment, and unavailability or interruption of operations.

6. DISCLAIMER OF LIABILITY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, NEITHER RECIPIENT NOR ANY CONTRIBUTORS SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE PROGRAM OR THE EXERCISE OF ANY RIGHTS GRANTED HEREUNDER, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. GENERAL

If any provision of this Agreement is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this Agreement, and without further action by the parties hereto, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

If Recipient institutes patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Program itself (excluding combinations of the Program with other software or hardware) infringes such Recipient's patent(s), then such Recipient's rights granted under Section 2(b) shall terminate as of the date such litigation is filed.

All Recipient's rights under this Agreement shall terminate if it fails to comply with any of the material terms or conditions of this Agreement and does not cure such failure in a reasonable period of time after becoming aware of such noncompliance. If all Recipient's rights under this Agreement terminate, Recipient agrees to cease use and distribution of the Program as soon as reasonably practicable. However, Recipient's obligations under this Agreement and any licenses granted by Recipient relating to the Program shall continue and survive.

Everyone is permitted to copy and distribute copies of this Agreement, but in order to avoid inconsistency the Agreement is copyrighted and may only be modified in the following manner. The Agreement Steward reserves the right to publish new versions (including revisions) of this Agreement from time to time. No one other than the Agreement Steward has the right to modify this Agreement. The Eclipse Foundation is the initial Agreement Steward. The Eclipse Foundation may assign the responsibility to serve as the Agreement Steward to a suitable separate entity. Each new version of the Agreement will be given a distinguishing version number. The Program (including Contributions) may always be distributed subject to the version of the Agreement under which it was received. In addition, after a new version of the Agreement is published, Contributor may elect to distribute the Program (including its Contributions) under the new version. Except as expressly stated in Sections 2(a) and 2(b) above, Recipient receives no rights or licenses to the

intellectual property of any Contributor under this Agreement, whether expressly, by implication, estoppel or otherwise. All rights in the Program not expressly granted under this Agreement are reserved.

This Agreement is governed by the laws of the State of New York and the intellectual property laws of the United States of America. No party to this Agreement will bring a legal action under this Agreement more than one year after the cause of action arose. Each party waives its rights to a jury trial in any resulting litigation.

Related Links

TrueLicense on page 640

GNU General Public License, version 2, with the Classpath Exception

The GNU General Public License (GPL)

Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software – to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means

either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

One line to give the program's name and a brief idea of what it does.

Copyright (C) <year> <name of author>

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) year name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items – whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989

Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

"CLASSPATH" EXCEPTION TO THE GPL Certain source files distributed by Oracle America and/or its affiliates are subject to the following clarification and special exception to the GPL, but only where Oracle has expressly included in the particular source file's header the words "Oracle designates this particular file as subject to the "Classpath" exception as provided by Oracle in the LICENSE file that accompanied this code."

Linking this library statically or dynamically with other modules is making a combined work based on this library. Thus, the terms and conditions of the GNU General Public License cover the whole combination.

As a special exception, the copyright holders of this library give you permission to link this library with independent modules to produce an executable, regardless of the license terms of these independent modules, and to copy and distribute the resulting executable under terms of your choice, provided that you also meet, for each linked independent module, the terms and conditions of the license of that module. An independent module is a module which is not derived from or based on this library. If you modify this library, you may extend this exception to your version of the library, but you are not obligated to do so. If you do not wish to do so, delete this exception statement from your version.

Related Links

IKVM on page 636

JSON License

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

The Software shall be used for Good, not Evil.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Related Links

JSON on page 637

The GNU Lesser General Public License, version 2.1 (LGPL-2.1)

GNU Lesser General Public License

Version 2.1, February 1999

Copyright (C) 1991, 1999 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public License, version 2, hence the version number 2.1.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software – to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software packages – typically libraries – of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) The modified work must itself be a software library.

b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through

that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Related Links

cron4j on page 634

GeoIP API on page 635

JCommon on page 637

The GNU Lesser General Public License, version 3.0 (LGPL-3.0)

This license is a set of additional permissions added to version 3 of the GNU General Public License.

GNU LESSER GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates the terms and conditions of version 3 of the GNU General Public License, supplemented by the additional permissions listed below.

0. Additional Definitions.

As used herein, “this License” refers to version 3 of the GNU Lesser General Public License, and the “GNU GPL” refers to version 3 of the GNU General Public License.

“The Library” refers to a covered work governed by this License, other than an Application or a Combined Work as defined below.

An “Application” is any work that makes use of an interface provided by the Library, but which is not otherwise based on the Library. Defining a subclass of a class defined by the Library is deemed a mode of using an interface provided by the Library.

A “Combined Work” is a work produced by combining or linking an Application with the Library. The particular version of the Library with which the Combined Work was made is also called the “Linked Version”.

The “Minimal Corresponding Source” for a Combined Work means the Corresponding Source for the Combined Work, excluding any source code for portions of the Combined Work that, considered in isolation, are based on the Application, and not on the Linked Version.

The “Corresponding Application Code” for a Combined Work means the object code and/or source code for the Application, including any data and utility programs needed for reproducing the Combined Work from the Application, but excluding the System Libraries of the Combined Work.

1. Exception to Section 3 of the GNU GPL.

You may convey a covered work under sections 3 and 4 of this License without being bound by section 3 of the GNU GPL.

2. Conveying Modified Versions.

If you modify a copy of the Library, and, in your modifications, a facility refers to a function or data to be supplied by an Application that uses the facility (other than as an argument passed when the facility is invoked), then you may convey a copy of the modified version:

a) under this License, provided that you make a good faith effort to ensure that, in the event an Application does not supply the function or data, the facility still operates, and performs whatever part of its purpose remains meaningful, or

b) under the GNU GPL, with none of the additional permissions of this License applicable to that copy.

3. Object Code Incorporating Material from Library Header Files.

The object code form of an Application may incorporate material from a header file that is part of the Library. You may convey such object code under terms of your choice, provided that, if the incorporated material is not limited to numerical parameters, data structure layouts and accessors, or small macros, inline functions and templates (ten or fewer lines in length), you do both of the following:

a) Give prominent notice with each copy of the object code that the Library is used in it and that the Library and its use are covered by this License.

b) Accompany the object code with a copy of the GNU GPL and this license document.

4. Combined Works.

You may convey a Combined Work under terms of your choice that, taken together, effectively do not restrict modification of the portions of the Library contained in the Combined Work and reverse engineering for debugging such modifications, if you also do each of the following:

a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.

b) Accompany the Combined Work with a copy of the GNU GPL and this license document.

c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.

d) Do one of the following:

0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.

1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.

e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

You may place library facilities that are a work based on the Library side by side in a single library together with other library facilities that are not Applications and are not covered by this License, and convey such a combined library under terms of your choice, if you do both of the following:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities, conveyed under the terms of this License.

b) Give prominent notice with the combined library that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

6. Revised Versions of the GNU Lesser General Public License.

The Free Software Foundation may publish revised and/or new versions of the GNU Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library as you received it specifies that a certain numbered version of the GNU Lesser General Public License “or any later version” applies to it, you have the option of following the terms and conditions either of that published version or of any later version published by the Free Software Foundation. If the Library as you received it does not specify a version number of the GNU Lesser General Public License, you may choose any version of the GNU Lesser General Public License ever published by the Free Software Foundation.

If the Library as you received it specifies that a proxy can decide whether future versions of the GNU Lesser General Public License shall apply, that proxy's public statement of acceptance of any version is permanent authorization for you to choose that version for the Library.

Related Links

FastColoredTextBox on page 635

The MIT License (MIT)

Copyright (c) <year> <copyright holders>

> **Note:** The copyright statement above is included in its completed form in the sections of this document specific to the individual products covered by this license.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Related Links

jQuery on page 637

Knockout on page 637

Rickshaw on page 638

SLF4J on page 639

About Push Technology

Founded in 2006 and headquartered in London, Push Technology is an innovative and global technology and solutions specialist that focuses on projecting data beyond the edge. Uniquely capable of delivering Data on Demand, Push Technology's robust and scalable communication platform – **Diffusion** - enables large-scale smart data distribution that unlocks the true potential of today's multi-channel, connected world to help organizations realise meaningful, valuable and personalized exchanges.

With offices in New York, Maidenhead and London, Push Technology has hundreds of man years' experience working on mission-critical data and messaging solutions for leading companies in the Financial Services, e-Gaming, Social Gaming and Digital Media Sectors. It is this expertise and leadership in delivering Data on Demand, coupled with a proven track-record in successful implementations that has earned Push Technology its hard-won reputation for cutting-edge, reliable, scalable and high performing solutions.

Working with Push Technology, organizations with demanding real-time two-way communications requirements – such as online gaming companies, financial institutions, brokers, traders, spread betting firms and news feed providers – can successfully redefine the boundaries of their Internet performance.

For further information
Visit **www.pushtechnology.com** or contact **salesinfo@pushtechnology.com**

PUSH TECHNOLOGY LIMITED

3RD FLOOR HOLLAND HOUSE
1-4 BURY STREET LONDON EC3A 5AW
Telephone: +44 (0)203 588 0900

340 MADISON AVE, NY, NY 10173
Telephone: +1 212 220 9287

Twitter: @push_technology
www.pushtechnology.com

© Copyright Push Technology Limited 2014